
Chapter 2. System Calls

About This Chapter

This chapter gives detailed information about each of the system calls that are available in the AIX Operating System. System calls provide controlled access to the operating system kernel.

The programming interface to the system calls is identical to that of subroutines. Thus, as far as a C-language program is concerned, a system call is merely a subroutine call. The real difference between a system call and a subroutine is the type of operation it performs. When a program invokes a system call, a context switch takes place so that the called routine has access to the operating system kernel's delicate information. The routine then operates in kernel mode to perform a task on behalf of the program. In this way, access to the delicate system information is restricted to a pre-defined set of routines whose actions can be controlled.

The operations performed by system calls are frequently more basic or "primitive" than those of subroutines. Many subroutines described in Chapter 3, "Subroutines," use system calls to perform more complex tasks. For example, the **open**, **close**, **read**, and **write**, system calls perform very simple I/O operations; but many programs use a standard set of I/O subroutines that add data buffering to the I/O performed by the system calls. (See "stdio" on page 3-297 for details about the Standard I/O Package.)

When an error occurs, most system calls return a value of -1 and set an external variable named **errno** to identify the error. The **errno.h** header file declares the **errno** variable and defines a constant for each of the possible error conditions. A complete listing of these error codes and their meanings can be found in Appendix A, "Error Codes." The specific meanings of the error codes that apply to each system call are listed in the "Diagnostics" section of each system call entry.

For an explanation of the "Syntax" section of each entry, see "Syntax" on page v. For an explanation of header files, see "Header Files" on page vi.

The following discussion is divided into sections that discuss groups of system calls that perform various operations.

Input/Output

The following system calls perform the basic input/output for all types of devices:

access	Determines whether the process has permission to access a file.
chdir	Changes the current directory.
close	Closes a file.
creat	Creates a new file or replaces an existing file with an empty one.
dup	Duplicates an open file descriptor.
fclear	Clears space in a file, freeing unused disk space.
fsync	Forces changes to a file to be written to the disk.
ftruncate	Shortens a file.
ioctl	Provides device-specific control.
lockf	Locks a region of a file from access by other processes.
lseek	Moves the read/write pointer of a file.
open	Opens a file or device for reading or writing.
read	Reads data from a file or device.
write	Writes data to a file or device.

File Maintenance

The file maintenance calls change the access permissions of files, create directories, mount file systems, and perform a variety of other operations:

chmod	Changes the access permission mode of a file.
chown	Changes the user and group that own a file.
chroot	Changes the directory considered to be the root directory.
fcntl	Provides file control.
fstat	Gets file status information.
link	Creates a new directory entry that links to a file.
mknod	Creates a special file that describes a device.
mount	Mounts a file system.
stat	Gets file status information.
sync	Forces all changes in the file system to be written to disk.
umask	Sets the file creation mask.
umount	Unmounts a file system.
unlink	Removes a directory entry.
ustat	Gets file system statistics.
utime	Set the access and modification times of a file.

Process Control

The following system calls control creating, operating and stopping processes:

exec	Replaces the current process image with a new program.
exit	Terminates the current process.
fork	Creates and starts a child process.
nice	Changes the execution priority of a process.
plock	Locks a process in memory.
wait	Waits for a child process to stop or terminate.

Process Identification

The following system calls get and set the IDs and limits of a process:

getegid	Gets the effective group ID.
geteuid	Gets the effective user ID.
getgid	Gets the real group ID.
getgroup	Gets the group access list.
getpggrp	Gets the process group ID.
getpid	Gets the process ID.
getppid	Gets parent process ID.
getuid	Gets the real user ID.
setgid	Sets the real and effective group IDs.
setgroups	Sets the group access list.
setpggrp	Sets the process group ID.
setuid	Sets the real and effective user IDs.
ulimit	Gets and sets the process's user limits.

Signals

Signals are sent to processes when exceptional events occur. A signal interrupts the activity that a process is performing and causes it to take a special action. For example, when a user presses the **Alt-Pause** key sequence at a work station, the **SIGINT** signal is sent to the user's processes. Normally, this causes them to terminate, but each process can arrange to ignore the signal, or to take some other action. The signals that can occur are defined in the **sys/signal.h** header file, and they are further described in "signal" on page 2-139.

Standard signal processing is compatible with UNIX System V and is described in more detail in "signal" on page 2-139. The following system calls handle standard signal processing:

alarm	Sets the process's alarm clock.
kill	Sends a signal to one or more processes.
pause	Suspends the process until a signal arrives.
signal	Sets the action to take when the process receives a signal.

Enhanced signal processing adds several useful features to the facility. It is described in more detail in "Signals" on page 1-20. The following system calls control enhanced signal processing:

sigblock	Blocks signals from being received.
sigsetmask	Sets the signal mask.
sigpause	Suspends the process until a signal arrives.
sigstack	Specifies an alternate stack upon which to process signals.
sigvec	Sets the action to take when the process receives a signal.

Semaphores, Message Queues, and Shared Memory Segments

In addition to signals, the AIX Operating System provides three facilities that provide flexible interprocess communication (IPC): semaphores, message queues, and shared memory segments. Details about the philosophy and use of each these facilities is beyond the scope of this book.

The names of the system calls that deal with semaphores begin with the letters **sem-**. The message queue system calls begin with **msg-**, and the shared memory system calls begin with **shm-**. All three facilities are accessed in a similar manner. The steps are outlined here in approximately the order that they appear in programs:

1. The user specifies a *key* to identify the individual semaphore set, message queue, or shared segment to be accessed. This key is analogous to a file name in that it has been previously agreed upon to identify a specific data structure.

The key **IPC_PRIVATE** (defined in the **sys/ipc.h** header file) is a special key value that specifies that the data structure is to be private to the current process.

Keys can be generated by any algorithm as long as the same algorithm is used by all processes on the system. The **ftok** subroutine provides a standard algorithm for generating IPC keys. (See "ftok" on page 3-183 for information about this subroutine.)

2. System calls whose names end with **-get** (**semget**, **msgget**, and **shmget**) use the key to obtain access to the requested data structure. The **-get** system calls are analogous to **open**: each returns an integer identifier (analogous to a file descriptor) that identifies the data structure for access with other system calls.

Normally, if the semaphore, message queue, or shared segment does not already exist, then the **-get** system call creates the necessary data structure. If another process has already created the data structure by calling the same **-get** system call with the same *key*, then the the identifier of that data structure is returned. This action can be modified with the **semflg**, **msgflg**, or **shmflg** parameter.

However, if **IPC_PRIVATE** is specified as the key, then a private data structure is created. No key exists with which to identify this data structure, so only processes that have its identifier can access it. The current process must pass the identifier to other processes that are to access it. For example, the identifier can be passed to a

child process through the *argv* argument vector (see “exec: *execl*, *execv*, *execle*, *execve*, *execlp*, *execvp*” on page 2-32 for details).

3. Shared memory segments must next be attached using the **shmat** system call.
4. The **semop** system call accesses semaphores. Message queues are accessed by **msgsnd**, **msgrcv**, and **msgxrcv**. Programs can access shared memory segments as regular memory through the pointer returned by the **shmat** system call.
5. System calls whose names end with **-ctl** (**semctl**, **msgctl**, and **shmctl**) perform a variety of control operations on the data structure. These control operations include getting status information and changing the access permissions. The data structure associated with each type of IPC identifier is defined in the description of the corresponding **-ctl** system call.
6. When no longer in use, shared memory segments must be detached using the **shmdt** system call.
7. The IPC identifier and the associated data structure should then be removed from the system with the **IPC_RMID** operation of the corresponding **-ctl** system call.

Each IPC data structure contains an **ipc_perm** structure, which contains access permission information. The **ipc_perm** structure is defined in the **sys/ipc.h** header file and it contains the following members:

```
ushort uid; /* Owner's user ID */
ushort gid; /* Owner's group ID */
ushort cuid; /* Creator's user ID */
ushort cgid; /* Creator's group ID */
ushort mode; /* Access permission mode */
ushort seq; /* Slot usage sequence number */
key_t key; /* Key */
```

The access permission mechanism resembles the one for files, except that execute permission does not exist for IPC facilities. The **semget**, **msgget**, and **shmget** system calls set the initial permissions when they create new IPC data structures. Also, the user (group) permissions apply if the process's effective user (group) ID matches either **uid** (**gid**) or **cuid** (**cgid**). The permissions can be changed with the corresponding **-ctl** system calls. The **uid** and **gid** fields identify the user and group that own the file for determining whether a given process may access a data structure. The **cuid** and **cgid** fields identify the process that created the data structure, and they can not be changed.

The **mode** field is constructed by logically OR-ing one or more of the following values. Note that these values are defined in the **sys/stat.h** header file and that they are a subset of the access permissions that apply to files.

S_IRUSR	Permits the process that owns the data structure to read it.
S_IWUSR	Permits the process that owns the data structure to modify it.
S_IRGRP	Permits the group associated with the data structure to read it.
S_IWGRP	Permits the group associated with the data structure to modify it.
S_IROTH	Permits others to read the data structure.
S_IWOTH	Permits others to modify the data structure.

When the individual IPC system calls are discussed in this chapter, the notation {READ} indicates that the current process must have read permission in order to perform the operation being described. Similarly, {WRITE} indicates that the process must have write permission to perform the operation.

For more information about the interprocess communication facilities, see *AIX Operating System Programming Tools and Interfaces*.

The **shmat** system call can be used to attach mapped files as well as shared memory segments. The two functions are similar in some ways, but they should not be confused. For an overview of the mapped file facility, see page 1-13. For more detailed information, see the following discussion and “shmat” on page 2-125.

Mapped Files

The AIX Operating System allows programs to map a file onto a memory segment so that that data in the file can be accessed more quickly and more directly. See page 1-13 for an overview of the file-mapping facility.

The steps for setting up and accessing a mapped file are outlined here in the order that they occur in programs:

1. Open the file with the **open** system call. The file must be a regular file. Directories and special files cannot be mapped.
2. Attach the file to a memory segment using the **shmat** system call. Specify the file descriptor returned by **open** in place of the shared memory identifier as the *shmid* parameter. The *shmflg* parameter is either **SHM_MAP** (to select file mapping), or it is constructed by logically OR-ing the value **SHM_MAP** with one of the following values:

SHM_RDONLY	Maps the file in read-only mode.
SHM_COPY	Maps the file in copy-on-write mode.

If neither **SHM_RDONLY** nor **SHM_COPY** is set, then the file is mapped in read-write mode. The *shmflg* parameter can also be logically OR-ed with the following value:

SHM-RND	Rounds the address specified by the <i>shmaddr</i> parameter to the next lowest segment boundary, if necessary.
----------------	---

The file must be opened for writing (in step 1) before it can be mapped read-write or copy-on-write.

-
3. Access shared memory segments as regular memory through the pointer returned by the **shmat** system call. The system performs the necessary read operations for you automatically. If the file is mapped read-write, then the system automatically writes to the file as well.
 4. If the file is mapped copy-on-write, then you must explicitly tell the system to update the file by using the **fsync** system call. If you never call **fsync**, then changes made to the mapped file in memory are never written to permanent storage.
 5. The **shmctl** system call can be used to get status information about the memory segment onto which the file is mapped.
 6. If you wish, you can use the **shmdt** system call to unmap the file and detach the memory segment, leaving the file open for conventional I/O.
 7. Close the file with the **close** system call. **close** automatically detaches the memory segment (unless you already did this in step 6).

Mapped files can be shared with other processes that map the file, or that use the conventional I/O system calls. All of these processes access the same shared memory segment, except for those that write to the file after mapping it in copy-on-write mode. Each process that maps a file copy-on-write gets a private mapped copy of the file when it first attempt to write to it.

Warning: Data may be lost if a process modifies a file that another process has mapped copy-on-write. When the latter process calls **fsync**, the changes made by the former process are overwritten.

access

Purpose

Determines the accessibility of a file.

Syntax

```
#include <unistd.h>
```

```
int access (path, amode)
```

```
char *path;
```

```
int amode;
```

Description

The **access** system call checks the accessibility of the file specified by the *path* parameter. The **access** system call checks the named file to see if the type of access specified by the *amode* parameter is permitted. The **access** system call uses the real user ID in place of the effective user ID and the real group ID in place of the effective group ID.

The bit pattern contained in *amode* is constructed by logically OR-ing the following values:

R_OK	Checks read permission.
W_OK	Checks write permission.
X_OK	Checks execute (search) permission.
F_OK	Checks to see if the file exists.

The owner of a file has access checked with respect to the owner read, write, and execute mode bits. Members of the file's group other than the owner have access checked with respect to the group mode bits. All others have access checked with respect to the other mode bits.

Return Value

If the requested access is permitted, a value of 0 is returned. If the requested access is denied, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

Access to the file is denied if one or more of the following are true:

ENOTDIR	A component of the path prefix is not a directory.
ENOENT	Read, write, or execute (search) permission is requested for a null path name.
ENOENT	The named file does not exist.
EACCES	Search permission is denied on a component of the path prefix.
EACCES	Permission bits of the file mode do not permit the requested access.
EROFS	Write access is requested for a file on a read-only file system.
ETXTBSY	Write access is requested for a pure procedure (shared text) file that is being executed.
EFAULT	The <i>path</i> parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “chmod” on page 2-18 and “stat, fstat” on page 2-153.

acct

Purpose

Enables and disables process accounting.

Syntax

```
int acct (path)  
char *path;
```

Description

The **acct** system call enables the accounting routine when the *path* parameter is nonzero and no errors occur during the system call. When the accounting routine is enabled, an accounting record is written on the accounting file named by the *path* parameter for each process that terminates. (For information about the accounting file, see “acct” on page 4-15)

You disable the accounting routine when the *path* parameter is 0 or **NULL** and no errors occur during the system call.

The effective user ID of the calling process must be superuser to use the **acct** system call.

Return Value

Upon successful completion, **acct** returns a value of 0. If **acct** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **acct** system call fails if one or more of the following are true:

EPERM	The effective user ID of the calling process is not superuser.
EBUSY	An attempt is made to enable accounting when it is already enabled.
ENOTDIR	A component of the <i>path</i> parameter is not a directory.
ENOENT	Any component of the accounting file's path name does not exist.
EACCES	Any component of the <i>path</i> parameter denies search permission.
EACCES	The file named by the <i>path</i> parameter is not an ordinary file.

acct

EACCES	Mode permission is denied for the named accounting file.
EISDIR	The named file is a directory.
EROFS	The named file resides on a read-only file system.
EFAULT	The <i>path</i> parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “exit, _exit” on page 2-38, “signal” on page 2-139, and “acct” on page 4-15.

The discussion of **acct** in *Using and Managing the AIX Operating System*.

alarm

Purpose

Sets a process's alarm clock.

Syntax

```
unsigned int alarm (sec)
unsigned int sec;
```

Description

The **alarm** system call instructs the calling process's alarm clock to send a **SIGALRM** signal to the calling process after the number of real-time seconds specified by the *sec* parameter have elapsed. (See "signal" on page 2-139 for more information about signals.)

The **alarm** system calls are not stacked. Successive **alarm** system calls reset the calling process's alarm clock.

If the *sec* parameter is 0, any previous alarm request is canceled.

Return Value

The **alarm** system call returns the amount of time previously remaining in the calling process's alarm clock.

Related Information

In this book: "pause" on page 2-92 and "signal" on page 2-139.

brk, sbrk

brk, sbrk

Purpose

Changes data segment space allocation.

Syntax

int brk (*endds*)
char **endds*;

char ***sbrk** (*incr*)
int *incr*;

Description

The **brk** and **sbrk** system calls dynamically change the amount of space allocated for the calling process's data segment. (For information about data segments, see "exec: execl, execlp, execlv, execlx, execve, execlp, execlv" on page 2-32.)

The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the current end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is initialized to 0. The break value can be automatically rounded up to a size appropriate for the memory management architecture.

The **brk** system call sets the break value to the value of the *endds* parameter and changes the allocated space accordingly.

The **sbrk** system call adds to the break value the number of bytes contained in the *incr* parameter and changes the allocated space accordingly. The *incr* parameter can be a negative number, in which case the amount of allocated space is decreased.

Return Value

Upon successful completion, the **brk** system call returns a value of 0, and the **sbrk** system call returns the old break value. If the **brk** or the **sbrk** system calls fail, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **brk** and the **sbrk** system calls fail and the allocated space remains unchanged if one or more of the following are true:

- ENOMEM** The requested change will allocate more space than is allowed by a system-imposed maximum. (For information on the system-imposed maximum on memory space, see “ulimit” on page 2-161.)
- ENOMEM** The requested change will set the break value to a value greater than or equal to the start address of any attached shared memory segment. (For information on shared memory operations, see “shmat” on page 2-125, “shmdt” on page 2-132, and “shmget” on page 2-134.)

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “shmat” on page 2-125, “shmdt” on page 2-132, and “ulimit” on page 2-161.

chdir

chdir

Purpose

Changes the current directory.

Syntax

```
int chdir (path)  
char *path;
```

Description

The **chdir** system call changes the current directory to the directory specified by the *path* parameter. The **current directory**, also called the **current working directory**, is the starting point of searches for path names that do not begin with a / (slash).

Return Value

Upon successful completion, the **chdir** system call returns a value of 0. If the **chdir** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **chdir** system call fails, and the current directory remains unchanged if one or more of the following are true:

- | | |
|----------------|--|
| ENOTDIR | A component of the <i>path</i> parameter is not a directory. |
| ENOENT | The named directory does not exist. |
| EACCES | Search permission is denied for any component of the <i>path</i> parameter. |
| EFAULT | The <i>path</i> parameter points to a location outside of the process's allocated address space. |

Related Information

In this book: “chroot” on page 2-23.

The **cd** command in *AIX Operating System Commands Reference*.

chmod

chmod

Purpose

Changes file access permissions.

Syntax

```
#include <sys/stat.h>
```

```
int chmod (path, mode)
char *path;
int mode;
```

Description

The **chmod** system call sets the access permissions of the file specified by the *path* parameter. The access permissions of the file are set according to the bit pattern specified by the *mode* parameter.

To change file access permissions, the effective user ID of the calling process must either be superuser or match the ID of the file's owner.

The *mode* parameter is constructed by logically OR-ing one or more of the following values, which are defined in the **sys/stat.h** header file:

S_ISUID	Sets the process's effective user ID to the file's owner on execution.
S_ISGID	Sets the process's effective group ID to the file's group on execution.
S_ISVTX	Saves text image after execution.
S_ENFMT	Enables enforcement-mode record locking.
S_IRUSR	Permits the file's owner to read it.
S_IWUSR	Permits the file's owner to write to it.
S_IXUSR	Permits the file's owner to execute it (or to search the directory).
S_IRGRP	Permits the file's group to read it.
S_IWGRP	Permits the file's group to write to it.
S_IXGRP	Permits the file's group to execute it (or to search the directory).
S_IROTH	Permits others to read the file.
S_IWOTH	Permits others to write to the file.
S_IXOTH	Permits others to execute the file (or to search the directory).

Other mode values exist that can be set with the **mknod** system call, but not with **chmod**. A complete list of the possible file mode values and other useful macros appears in "stat.h" on page 5-61.

Setting **S_ISVTX** for a shared executable file prevents the system from unmapping the program text segment of the file when its last user terminates. Thus, when the next process executes it, the text need not be read from the file system. It is simply paged in, saving time.

If **S_ENFMT** is set and no execute permissions are set, then locks placed on the file with the **lockf** system call are **enforced locks**. See “lockf” on page 2-62 for details about locking regions of a file.

If the effective user ID of the calling process is not superuser, then the **chmod** system call clears the **S_ISVTX** bit.

If the effective user ID of the process is not that of superuser, or the effective group ID of the process does not match the file’s existing group ID, then the **chmod** system call clears the **S_ISGID** bit.

Return Value

Upon successful completion, the **chmod** system call returns a value of 0. If the **chmod** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Diagnostics

The **chmod** system call fails and the file permissions remain unchanged if one or more of the following are true:

- ENOTDIR** A component of the *path* parameter is not a directory.
- ENOENT** The named file does not exist.
- EACCES** A component of the *path* parameter has search permission denied.
- EPERM** The effective user ID does not match the ID of the owner of the file or the ID of superuser.
- EROFS** The named file resides on a read-only file system.
- EFAULT** The *path* parameter points to a location outside of the process’s allocated address space.

chmod

Related Information

In this book: “chown” on page 2-21 and “mknod” on page 2-67.

The **chmod** command in *AIX Operating System Commands Reference*.

chown

Purpose

Changes the owner and group IDs of a file.

Syntax

```
int chown (path, owner, group)  
char *path;  
int owner, group;
```

Description

The **chown** system call changes the owner ID and the group ID of the file named by the *path* parameter. The owner and group IDs of the named file are set to the numeric values contained in the *owner* and *group* parameters, respectively.

A process can change the ownership of a file only if its effective user ID is either superuser or the same as the file's owner ID.

If the effective user ID of the calling process is not the same as the superuser ID, then the **chown** system call clears the set-user-ID (**S_ISUID**) and set-group-ID (**S_ISGID**) bits of the file mode. (See "stat.h" on page 5-61 for the definitions of the constants **S_ISUID** and **S_ISGID**.)

Return Value

Upon successful completion, a value of 0 is returned. If the **chown** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **chown** system call fails and the owner ID and the group ID of the named file remain unchanged if one or more of the following are true:

- ENOTDIR** A component of the path prefix is not a directory.
- ENOENT** The named file does not exist.
- EACCES** Search permission is denied on a component of the path prefix.

chown

EPERM	The effective user ID does not match the owner of the file and the effective user ID is not superuser.
EROFS	The named file resides on a read-only file system.
EFAULT	The <i>path</i> parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “chmod” on page 2-18.

The **chown** command in *AIX Operating System Commands Reference*.

chroot

Purpose

Changes the effective root directory.

Syntax

```
int chroot (path)  
char *path;
```

Description

The **chroot** system call causes the directory named by the *path* parameter to become the effective root directory. The **effective root directory** is the starting point when searching for a file whose path name begins with /. The current directory is not affected by the **chroot** system call.

The effective user ID of the calling process must be superuser to change the effective root directory.

The .. (dot-dot) entry in the effective root directory is interpreted to mean the effective root directory itself. Thus, .. (dot-dot) cannot be used to access files outside the subtree rooted at the effective root directory.

Return Value

Upon successful completion, a value of 0 is returned. If the **chroot** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **chroot** system call fails and the effective root directory remains unchanged if one or more of the following are true:

- | | |
|----------------|--|
| ENOTDIR | Any component of the path name is not a directory. |
| ENOENT | The named directory does not exist. |
| EPERM | The effective user ID of the calling process is not superuser. |
| EFAULT | The <i>path</i> parameter points to a location outside of the process's allocated address space. |

chroot

Related Information

In this book: “chdir” on page 2-16.

The **chroot** command in *AIX Operating System Commands Reference*.

close

Purpose

Closes the file associated with a file descriptor.

Syntax

```
int close (fildes)  
int fildes;
```

Description

The **close** system call closes the file associated with the file descriptor *fildes*. The *fildes* parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call.

All file regions that this process has previously locked with the **lockf** system call are unlocked. This includes regions of files other than the file specified by the *fildes* parameter.

If the *fildes* parameter is associated with a mapped file, and if no other process has attached this mapped file, then it is unmapped.

Return Value

Upon successful completion, a value of 0 is returned. If the **close** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **close** system call fails if the following is true:

EBADF The *fildes* parameter is not a valid open file descriptor.

close

Related Information

In this book: “creat” on page 2-27, “dup” on page 2-30, “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “fcntl” on page 2-42, “open” on page 2-88, and “pipe” on page 2-93.

creat

Purpose

Creates a new file or rewrites an existing file.

Syntax

```
#include <stat.h>

int creat (path, mode)
char *path;
int mode;
```

Description

The **creat** system call creates a new ordinary file or prepares to rewrite an existing file named by the *path* parameter.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. If the file does not exist, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of the *mode* parameter modified as follows:

- All bits set in the process's file mode creation mask are cleared. (For information about the file mode creation mask, see "umask" on page 2-163.)
- The **save text image after execution bit** of the file mode (**S_ISVTX**) is cleared. (For more information about this bit, see "chmod" on page 2-18.)

See "chmod" on page 2-18 for a detailed explanation of file modes.

No process can have more than 200 files open simultaneously. A new file can be created with a mode that forbids writing.

Note that the following two system calls are equivalent:

```
creat (path, mode)
```

```
open (path, O_WRONLY | O_CREAT | O_TRUNC, mode)
```

See "open" on page 2-88 for details about the **open** system call.

Return Value

Upon successful completion, a file descriptor (a nonnegative integer) is returned and the file is opened for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across **exec** system calls. (For information about control of open files, see “fcntl” on page 2-42.)

If the **creat** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **creat** system call fails if one or more of the following are true:

ENOTDIR	A component of the path prefix is not a directory.
ENOENT	A component of the path prefix does not exist.
EACCES	Search permission is denied on a component of the path prefix.
ENOENT	The path name is null.
EACCES	The file does not exist and the directory in which the file is to be created does not permit writing.
EROFS	The named file resides or would reside on a read-only file system.
ETXTBSY	The file is a pure procedure (shared text) file that is being executed.
EACCES	The file exists and write permission is denied.
EISDIR	The named file is an existing directory.
EMFILE	Two hundred (200) file descriptors are currently open.
EFAULT	The <i>path</i> parameter points to a location outside of the process's allocated address space.
ENFILE	The system file table is full.
EAGAIN	The named file contains a record lock owned by another process. See “lockf” on page 2-62 for information about record locks.

Related Information

In this book: “chmod” on page 2-18, “close” on page 2-25, “dup” on page 2-30, “lseek” on page 2-65 “open” on page 2-88, “read, readx” on page 2-104, “umask” on page 2-163, and “write, writex” on page 2-178.

dup

dup

Purpose

Duplicates an open file descriptor.

Syntax

```
int dup (fildes)  
int fildes;
```

Description

The **dup** system call returns a new file descriptor for the file descriptor pointed to by the *fildes* parameter. The *fildes* parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call. The **dup** system call returns a new file descriptor having the following in common with the original:

- The same open file or pipe
- The same file pointer (that is, both file descriptors share one file pointer)
- The same access mode (read, write or read/write).

The new file descriptor is set to remain open across **exec** system calls. (For more information about file control, see “fcntl.h” on page 5-48.)

The file descriptor returned is the lowest one available.

Return Value

Upon successful completion, a file descriptor (nonnegative integer) is returned. If the **dup** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **dup** system call fails if one or more of the following are true:

- | | |
|---------------|--|
| EBADF | <i>fildes</i> is not a valid open file descriptor. |
| EMFILE | Two hundred (200) file descriptors are currently open. |

Related Information

In this book: “creat” on page 2-27, “close” on page 2-25, “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “fcntl.h” on page 5-48, “open” on page 2-88, and “pipe” on page 2-93.

exec: execl, execv, execl, execve, execlp, execvp

exec: execl, execv, execl, execve, execlp, execvp

Purpose

Executes a file.

Syntax

```
int execl (path, arg0 [, arg1, . . . ], 0)
char *path, *arg0, *arg1, . . . ;
```

```
int execl (path, arg0 [, arg1, . . . ], 0, envp)
char *path, *arg0, *arg1, . . . , *envp[ ];
```

```
int execlp (file, arg0 [, arg1, . . . ], 0)
char *file, *arg0, *arg1, . . . ;
```

```
int execv (path, argv)
char *path, *argv[ ];
```

```
int execve (path, argv, envp)
char *path, *argv[ ], *envp[ ];
```

```
int execvp (file, argv)
char *file, *argv[ ];
```

Description

The **exec** system call, in all its forms, executes a new program in the calling process. **exec** does not create a new program, but overlays the current program with a new one, which is called the **new process image**. The new process image file can be one of three file types:

- An executable binary file in **a.out** format (see “a.out” on page 4-5)
- An executable text file that contains a shell procedure (only **execlp** and **execvp** allow this type of new process image file)
- A file that names an executable binary file or shell procedure to be run.

The last of the types mentioned is recognized by a header with the syntax:

```
#! path [string]
```

The **#!** is the file's **magic number**, which identifies the file type. *path* is the path name of the file to be executed. *string* is an optional character string that contains no tab or space characters. If specified, this string is passed to the new process as an argument in front of the name of the new process image file. The header must be terminated with a new-line character. When invoked, the new process is passed *path* as *argv*[0]. If a *string* is specified in the new process image file, then the **exec** system call sets *argv*[0] to *string* and *path* concatenated together. The rest of the arguments passed are the same as those passed to the **exec** system call.

exec: execl, execv, execl, execve, execlp, execvp

The parameters for the **exec** system calls are defined as follows:

- path* This parameter points to the path name of the new process image file.
- file* This parameter points to the name of the new process image file. Unless *file* is a full path name, the path prefix for the file is obtained by searching the directories named in the **PATH** environment variable. The initial environment is supplied by the shell.

Note that **execlp** and **execvp** take *file* parameters, but the rest of the **exec** system calls take *path* parameters. (For information about the environment, see “environment” on page 5-44 and the **sh** command in *AIX Operating System Commands Reference*.)

arg0 [, *arg1*, . . .]

These parameters point to null-terminated character strings. The strings constitute the argument list available to the new process. By convention, at least *arg0* must be present, and it must point to a string that is the same as *path* or its last component.

argv This parameter is an array of pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one element, and it must point to a string that is the same as *path* or its last component. The last element of *argv* is a **NULL** pointer.

envp This parameter is an array of pointers to null-terminated character strings. These strings constitute the **environment** for the new process. The last element of *envp* is a **NULL** pointer.

When a C program is executed, it receives the following parameters:

```
main (argc, argv, envp)
int argc;
char *argv[ ], *envp[ ];
```

Here *argc* is the argument count, and *argv* is an array of character pointers to the arguments themselves. By convention, the value of *argc* is at least one, and *argv*[0] points to a string containing the name of the new process image file.

The **main** routine of a C language program automatically begins with a run-time start-off routine. This routine sets a global variable named **environ** so that it points to the environment array passed to the program in **envp**. You can access this global variable by including the following declaration in your program:

```
extern char **environ;
```

The **execl**, **execv**, **execlp**, and **execvp** system calls use **environ** to pass the calling process's current environment to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose **close-on-exec** flag is set. For those file descriptors that remain open, the file pointer is unchanged. (For information about file control, see “fcntl” on page 2-42.)

exec: execl, execv, execl, execve, execlp, execvp

If the new process requires shared libraries, **exec** will find, open, and map each shared library image to the new process address space. (See *AIX Operating System Programming Tools and Interfaces*.)

The **exec** system calls reset all caught signals to the default action. Signals that cause the default action continue to do so after **exec**. Ignored signals remain ignored, the signal mask remains the same, and the signal stack state is reset. (For information about signals, see “signal” on page 2-139 and “sigvec” on page 2-150.)

If the **set-user-ID** mode bit of the new process image file is set, then **exec** sets the effective user ID of the new process to the owner ID of the new process image file. Similarly, if the **set-group-ID** mode bit of the new process image file is set, then the effective group ID of the new process is set to the group ID of the new process image file. The real user ID and real group ID of the new process remain the same as those of the calling process. (For information about the set-ID modes, see “chmod” on page 2-18.)

The shared libraries attached to the calling process are not attached to the new process. (For information about shared memory segments, see “shmat” on page 2-125, “shmdt” on page 2-132, and “shmget” on page 2-134.)

Profiling is disabled for the new process. (For information about profiling, see “profil” on page 2-97.)

The new process inherits the following attributes from the calling process:

- Nice value (see “nice” on page 2-86)
- Process ID
- Parent process ID
- Process group ID
- semadj values (see “semop” on page 2-116)
- TTY group ID (see “exit, _exit” on page 2-38 and “signal” on page 2-139)
- Trace flag (see request 0 of “ptrace” on page 2-100)
- Time left until an alarm clock signal (see “alarm” on page 2-13)
- Current directory
- Root directory
- File mode creation mask (see “umask” on page 2-163)
- File size limit (see “ulimit” on page 2-161)
- **utime**, **stime**, **ctime**, and **cstime** (see “times” on page 2-159).

Return Value

Upon successful completion, **exec** does not return because the calling process image is overlaid by the new process image. If **exec** returns to the calling process, then it returns the value -1 and sets **errno** to indicate the error.

Diagnostics

The **exec** system call fails and returns to the calling process if one or more of the following are true:

- | | |
|----------------|--|
| ENOENT | One or more components of the new process image file's path name do not exist. |
| ENOTDIR | A component of the path prefix of the new process image file is not a directory. |
| EACCES | Search permission is denied for a directory listed in the path prefix of the new process image file. |
| EACCES | The new process image file is not an ordinary file. |
| EACCES | The mode of the new process image file denies execution permission. |
| ENOEXEC | The exec is not an execlp or execvp , and the new process image file has the appropriate access permission but has an invalid magic number in its header. |
| EINVAL | The new process image file has a valid magic number in its header, but the header is damaged or is incorrect for the machine on which the file is to be run. |
| ETXTBSY | The new process image file is a pure procedure (shared text) file that is currently open for writing by some process. |
| ENOMEM | The new process requires more memory than is allowed by the system-imposed maximum MAXMEM . |
| E2BIG | The number of bytes in the new process's argument list is greater than the system-imposed limit. This limit is defined as NCARGS in the sys/param.h header file. |
| EFAULT | The <i>path</i> , <i>argv</i> , or <i>envp</i> parameter points to a location outside of the process's allocated address space. |

In addition, some errors can occur when using the new process file after the old process image has been overwritten. These errors include problems in setting up new data and stack registers, problems in mapping a shared library, or problems in reading the new process file. Because returning to the calling process is not possible, the system sends the **SIGKILL** signal to the process when one of these errors occurs.

exec: execl, execv, execl, execve, execlp, execvp

If an error occurred while mapping a shared library, an error message describing the reason for failure will be written to standard error before the signal **SIGKILL** is sent to the process. (See *AIX Operating System Programming Tools and Interfaces*.) If a shared library cannot be mapped, one or more of the following is true:

ENOENT	One or more components of the path name of the shared library file do not exist.
ENOTDIR	A component of the path prefix of the shared library file is not a directory.
EACCES	Search permission is denied for a directory listed in the path prefix of the shared library file.
EACCES	The shared library file mode denies execution permission.
ENOEXEC	The shared library file has the appropriate access permission but an invalid magic number in its header.
ETXTBSY	The shared library file is currently open for writing by some other process.
ENOMEM	The shared library requires more memory than is allowed by the system-imposed maximum.

Example

The following is an example of a new process file that names a program to be run:

```
#!/usr/bin/awk -f
{ for (i = NF; i > 0; --i) print i }
```

If this file is named `reverse`, then typing the following command on the command line:

```
reverse chapter1 chapter2
```

causes the following command to be run:

```
/usr/bin/awk -f reverse chapter1 chapter2
```

Note that the **exec** system calls use only the first line of the new process image file and ignore the rest of it. Also, **awk** interprets the text that follows a **#** (number sign) as a comment. (See the **awk** command in *AIX Operating System Commands Reference* for more information.)

Related Information

In this book: “alarm” on page 2-13, “chmod” on page 2-18, “exit, _exit” on page 2-38, “fcntl” on page 2-42, “fork” on page 2-44, “nice” on page 2-86, “profil” on page 2-97, “ptrace” on page 2-100, “semop” on page 2-116, “shmat” on page 2-125, “signal” on page 2-139, “sigvec” on page 2-150, “times” on page 2-159, “ulimit” on page 2-161, “umask” on page 2-163, “varargs” on page 3-322, “a.out” on page 4-5, and “environment” on page 5-44.

The **sh** and **shlib** commands in *AIX Operating System Commands Reference*.

exit, _exit

exit, _exit

Purpose

Terminates a process.

Syntax

```
void exit (status)
int status;
```

```
void _exit (status)
int status;
```

Description

The **exit** system call terminates the calling process and causes the following to occur:

- All of the file descriptors open in the calling process are closed.
- If the parent process of the calling process is executing a **wait** system call, it is notified of the termination of the calling process and the low-order eight bits (that is, bits 0377 or 0xFF) of *status* are made available to it. See “wait” on page 2-176.
- If the parent process of the calling process is not executing a **wait** system call, and if the parent hasn’t set its **SIGCLD** signal to **SIG_IGN**, then the calling process is transformed into a zombie process. A **zombie process** is a process that occupies a slot in the process table, but has no other space allocated to it either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information to be used by the **times** system call. (See “times” on page 2-159 and the **sys/proc.h** header file.)
- The parent process ID of all of the calling process’s existing child processes and zombie processes is set to 1. This means the initialization process inherits each of these processes.
- Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.
- For each semaphore for which the calling process has set a **semadj** value, that **semadj** value is added to the **semval** of the specified semaphore. (See “semop” on page 2-116 about semaphore operations.)
- If the process has a process lock, text lock, or data lock, an **unlock** is performed. (See “plock” on page 2-95.)

- An accounting record is written on the accounting file if the system's accounting routine is enabled. (See "acct" on page 2-11 for information about enabling accounting routines.)
- If the process ID, tty group ID, and process group ID of the calling process are equal, then the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process. In other words, if **exit** is called by the process group leader for the controlling terminal (typically the shell), then **SIGHUP** is sent to all of the processes associated with that terminal.
- Locks set by the **lockf** system call are removed. (See "lockf" on page 2-62 about file locks.)

The **exit** subroutine causes cleanup actions to occur before the process exits. The **_exit** system call bypasses all cleanup.

Note: The effect of **exit** can be modified by the setting of the **SIGCLD** signal in the parent process. See "signal" on page 2-139 and "sigvec" on page 2-150.

Related Information

In this book: "acct" on page 2-11, "signal" on page 2-139, "sigvec" on page 2-150, "times" on page 2-159, and "wait" on page 2-176.

fclear

fclear

Purpose

Makes a hole in a file.

Syntax

```
long fclear (fildes, nbytes)  
int fildes;  
unsigned long nbytes;
```

Description

The **fclear** system call zeroes the number of bytes specified by the *nbytes* parameter starting at the current position of the file open on file descriptor *fildes*. This function differs from the logically equivalent write operation in that it returns full blocks of binary zeros to the file system, constructing **holes** in the file. The seek pointer of the file is advanced by *nbytes*.

If you **fclear** past the end of a file, then rest of the file is cleared and the seek pointer is advanced by *nbytes*. The file size is updated to include this new hole, which leaves the current file position at the byte immediately beyond the new end-of-file.

Return Value

Upon successful completion, a value of *nbytes* is returned. If the **fclear** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **fclear** system call fails if one or more of the following are true:

- | | |
|---------------|---|
| EIO | I/O error. |
| EBADF | The <i>fildes</i> option is not a valid file descriptor open for writing. |
| EINVAL | The file is a FIFO, directory, or special file. |
| EMFILE | The file is mapped copy-on-write by one or more processes. |

Related Information

In this book: “ftruncate” on page 2-48.

fcntl

fcntl

Purpose

Controls open file descriptors.

Syntax

```
#include <fcntl.h>
```

```
int fcntl (fildes, cmd, arg)  
int fildes, cmd, arg;
```

Description

The **fcntl** system call performs controlling operations on open file descriptors.

The *fildes* parameter is an open file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call. The *arg* parameter is variable that depends on value of the *cmd* parameter.

The *cmd* parameter specifies the operation to be performed, and it is one of the following:

- F_DUPFD** Returns a new file descriptor as follows:
- Lowest numbered available file descriptor greater than or equal to *arg*.
 - Same open file (or pipe) as the original file
 - Same file pointer as the original file (that is, both file descriptors share one file pointer)
 - Same access mode (read, write or read/write)
 - Same file status flags (that is, both file descriptors share the same file status flags)
 - The close-on-exec flag associated with the new file descriptor is set to remain open across **exec** system calls.
- F_GETFD** Gets the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is 0 (zero), then the file remains open across **exec** system calls; otherwise the file closes upon execution of an **exec** system call.
- F_SETFD** Sets the close-on-exec flag associated with the *fildes* parameter to the value of the low-order bit of *arg* (0 or 1 as for **F_GETFD**).

- F_GETFL** Gets the file status flags of the file descriptor *fildev*.
- F_SETFL** Sets the file status flags to the value of the *arg* parameter. Only the flags **O_NDELAY** and **O_APPEND** should be set. Attempting to set other flags may cause unexpected results.

Return Value

Upon successful completion, the value returned depends on the value of the *cmd* parameter as follows:

<i>cmd</i>	Return Value
F_DUPFD	A new file descriptor
F_GETFD	The value of the flag (only the low-order bit is defined)
F_SETFD	A value other than -1
F_GETFL	The value of file flags
F_SETFL	A value other than -1.

If the **fcntl** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **fcntl** system call fails if one or more of the following are true:

- EBADF** The *fildev* parameter is not a valid open file descriptor.
- EMFILE** The *cmd* parameter is **F_DUPFD** and 200 file descriptors are currently open.
- EINVAL** The *cmd* parameter is **F_DUPFD** and the *arg* parameter is negative or greater than 199.

Related Information

In this book: “close” on page 2-25, “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “open” on page 2-88, and “fcntl.h” on page 5-48.

fork

fork

Purpose

Creates a new process.

Syntax

```
int fork ( )
```

Description

The **fork** system call creates a new process. The new process (child process) is an exact copy of the calling process (parent process). The created child process inherits the following attributes from the parent process:

- Environment
- Close-on-exec flags (see “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32)
- Signal handling settings (that is, **SIG_DFL**, **SIG_IGN**, *function address*)
- Set-user-ID mode bit
- Set-group-ID mode bit
- Profiling on/off status
- Nice value (see “nice” on page 2-86)
- All attached shared libraries (see **shlib** command in *AIX Operating System Commands Reference*)
- Process group ID
- TTY group ID (see “exit, _exit” on page 2-38 and “signal” on page 2-139)
- Time left until an alarm clock signal (see “alarm” on page 2-13)
- Current directory
- Root directory
- File mode creation mask (see “umask” on page 2-163)
- File size limit (see “ulimit” on page 2-161)
- Attached shared memory segments (see “shmat” on page 2-125)
- Attached mapped file segments (see “shmat” on page 2-125).

The child process differs from the parent process in the following ways:

- The child process has a unique process ID.
- The child process has as its parent process ID the process ID of the parent process.

- The child process has its own copy of the parent's file descriptors. However, each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent process.
- All **semadj** values are cleared. (For information about **semadj** values, see "semop" on page 2-116.)
- Process locks, text locks and data locks are not inherited by the child. (For information about locks, see "plock" on page 2-95.)
- The child process's trace flag (see the discussion of request 0 of "ptrace" on page 2-100) is false regardless of the value of the parent process's trace flag.
- The child process's **utime**, **stime**, **cutime**, and **cstime** are set to 0. (See "times" on page 2-159.)

Return Value

Upon successful completion, **fork** returns a value of 0 to the child process and returns the process ID of the child process to the parent process. If **fork** fails, a value of -1 is returned to the parent process, no child process is created, and **errno** is set to indicate the error.

Diagnostics

The **fork** system call fails if one or more of the following are true:

- | | |
|---------------|--|
| EAGAIN | The system-imposed limit on the total number of processes executing would be exceeded. |
| EAGAIN | The system-imposed limit on the total number of processes executing for a single user would be exceeded. |
| ENOMEM | There is not enough space left for this process. |

Related Information

In this book: "exec: execl, execv, execl, execve, execlp, execvp" on page 2-32, "exit, _exit" on page 2-38, "nice" on page 2-86, "plock" on page 2-95, "ptrace" on page 2-100, "semop" on page 2-116, "shmat" on page 2-125, "signal" on page 2-139, "sigvec" on page 2-150, "times" on page 2-159, "ulimit" on page 2-161, "umask" on page 2-163, and "wait" on page 2-176.

The **shlib** command in *AIX Operating System Commands Reference*.

fsync

fsync

Purpose

Writes changes in a file to permanent storage.

Syntax

```
int fsync (fildes)  
int fildes;
```

Description

The **fsync** system call causes all modified data in the file open on *fildes* to be saved to permanent storage. If the file is mapped onto a segment in **read-write** mode, then it is saved to permanent storage. If the file is mapped **copy-on-write**, then the pages of the file that have been changed are saved to permanent storage. Saving to permanent storage is sometimes called a **commit operation**.

Return Value

Upon successful completion, **fsync** returns a value of 0. If **fsync** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **fsync** system call fails if one or more of the following are true:

- | | |
|---------------|--|
| EIO | I/O error. |
| EBADF | <i>fildes</i> is not a valid file descriptor open for writing. |
| EINVAL | The file is a FIFO file, directory, or special file. |

Related Information

In this book: “sync” on page 2-157.

ftruncate

ftruncate

Purpose

Makes a file shorter.

Syntax

```
int ftruncate (fildes, length)  
int fildes;  
unsigned long length;
```

Description

The **ftruncate** system call removes all data beyond *length* bytes from the beginning of the file that is open on the file descriptor *fildes*. Full blocks are returned to the file system so that they can be used again, and the file size is changed to the value of the *length* parameter.

The **ftruncate** subroutine does not modify the file's seek pointer.

Return Value

Upon successful completion, **ftruncate** returns a value of 0. If **ftruncate** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **ftruncate** system call fails if one or more of the following are true:

- | | |
|---------------|--|
| EIO | I/O error. |
| EBADF | <i>fildes</i> is not a valid file descriptor open for writing. |
| EINVAL | The file is a directory, FIFO, or special file. |
| EMFILE | The file is mapped copy-on-write by one or more processes. |

Related Information

In this book: “fclear” on page 2-40.

getgroups

getgroups

Purpose

Gets the group access list.

Syntax

```
#include <grp.h>
```

```
int getgroups (ngroups, gidset)  
int ngroups, *gidset;
```

Description

The **getgroups** system call gets the current group access list of the user process. The list is stored in the array pointed to by the *gidset* parameter. The *ngroups* parameter indicates the number of entries that can be stored in this array. **getgroups** never returns more than **NGROUPS** entries. (**NGROUPS** is a constant defined in the **grp.h** header file.)

Return Value

Upon successful completion, the **getgroups** system call returns the number of elements stored into the array pointed to by the *gidset* parameter. If **getgroups** fails, then a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **getgroups** system call fails if the following is true:

EFAULT The *ngroups* and *gidset* parameters specify an array that is partially or completely outside of the process's allocated address space.

Related Information

In this book: “setgroups” on page 2-120 and “initgroups” on page 3-214.

getpid, getpgrp, getppid

getpid, getpgrp, getppid

Purpose

Gets the process, process group, and parent process IDs.

Syntax

int getpid ()

int getpgrp ()

int getppid ()

Description

The **getpid** system call returns the process ID of the calling process.

The **getpgrp** system call returns the process group ID of the calling process.

The **getppid** system call returns the process ID of the calling process's parent process.

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “fork” on page 2-44, “setpgrp” on page 2-122, and “signal” on page 2-139.

getuid, geteuid, getgid, getegid

Purpose

Gets the real user, effective user, real group, and effective group IDs.

Syntax

unsigned short getuid ()

unsigned short getgid ()

unsigned short geteuid ()

unsigned short getegid ()

Description

The **getuid** system call returns the real user ID of the calling process.

The **geteuid** system call returns the effective user ID of the calling process.

The **getgid** system call returns the real group ID of the calling process.

The **getegid** system call returns the effective group ID of the calling process.

Related Information

In this book: “setuid, setgid” on page 2-123.

ioctl

ioctl

Purpose

Controls input/output devices.

Syntax

```
#include <sys/ioctl.h >
#include <sys/devinfo.h >

int ioctl (fildes, request, arg)
int fildes, request;
char *arg;
```

Description

The **ioctl** system call performs a variety of functions on both block and character special files (devices). These functions, and how **ioctl** applies to them, are described in Chapter 6, “Special Files” on page 6-1.

There are two requests that may be used in the *request* parameter for any special file that supports the **ioctl** system call. These two requests are:

IOCTYPE Returns the device type associated with *fildes*. The device types are defined in the **sys/devinfo.h** header file, which is discussed in “devinfo” on page 4-55.

IOCINFO Stores device information for the file specified by *fildes* into the buffer pointed to by the *arg* parameter. See “devinfo” on page 4-55 for the format of the device information structure.

Some devices support additional requests. See the discussion of particular devices in Chapter 6, “Special Files” on page 6-1 for details about device-dependent **ioctl** calls.

Return Value

If **ioctl** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **ioctl** fails if one or more of the following are true:

- EBADF** *fildev* is not a valid open file descriptor.
- ENOTTY** *fildev* is not associated with a character special file.
- ENODEV** The device associated with *fildev* does not support the **ioctl** system call.
- EFAULT** The *arg* parameter points to a location outside of the process's allocated address space.
- EINVAL** *request* or *arg* is not valid.
- EINTR** A signal was caught during the **ioctl** system call.

Related Information

In this book: “devinfo” on page 4-55 and Chapter 6, “Special Files.”

The discussion of **termio** in *AIX Operating System Programming Tools and Interfaces*.

iplvm, waitvm

iplvm, waitvm

Purpose

Starts a virtual machine or waits for one to terminate.

Syntax

int iplvm (*iodn*, *waitflag*)
unsigned short *iodn*;
int *waitflag*;

int waitvm (*iodn*)
unsigned short *iodn*;

Description

The **iplvm** system call starts (IPLs) a new virtual machine that is independent of the AIX virtual machine. The virtual machine is loaded from the device specified by the input/output device number (IODN) given in the *iodn* parameter.

If *iodn* is 0, then the virtual machine of the calling process is restarted (re-IPLed). This is accomplished by sending a **SIGQUIT** signal to the process 1. (Process 1 is also called the **init** process. See “Creation and Execution” on page 1-16 for more information about this special process.) Note that, unlike the **reboot** system call, **iplvm** performs a **sync** operation and writes all pending output to disk before restarting the virtual machine.

If the *waitflag* parameter is a nonzero value, then the **iplvm** system call waits until the new virtual machine has started before returning to the calling process. This allows you to determine whether the virtual machine IPLed successfully. *waitflag* is ignored if the *iodn* parameter is 0.

The **waitvm** system call waits for the virtual machine that was IPLed from the device specified by *iodn* to perform a virtual machine halt.

The calling process must have an effective user ID of superuser to perform either system call.

Return Value

Upon successful completion, the **iplvm** and **waitvm** system calls return a value of 0. If **iplvm** or **waitvm** fails, then a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **iplvm** and **waitvm** system calls fail if one or more of the following is true:

EPERM	The effective user ID of the calling process is not superuser.
EIO	The VRM detected one of the following error conditions: <ul style="list-style-type: none">• Insufficient resources are available.• The IPL diskette needs to be mounted.• The IPL header is not valid.• Error encountered while reading IPL record.• Duplicate virtual machine ID.• IPL key sequence error.
ENXIO	The device specified by the <i>iodn</i> parameter does not exist.
EINVAL	The <i>iodn</i> parameter is not valid.
EAGAIN	The maximum number of virtual machines are already running.

Related Information

In this book: “Creation and Execution” on page 1-16, “reboot” on page 2-107, “signal” on page 2-139, and “sync” on page 2-157.

The **init** command in *AIX Operating System Commands Reference*.

kill

kill

Purpose

Sends a signal to a process or to a group of processes.

Syntax

```
int kill (pid, sig)  
int pid, sig;
```

Description

The **kill** system call sends the signal specified by the *sig* parameter to the process or group of processes specified by the *pid* parameter. (For information on valid signals, see “signal” on page 2-139.) If the *sig* parameter is 0 (the **null signal**), error checking is performed but no signal is sent. This can be used to check the validity of *pid*.

To send a signal to another process, at least one of the following must be true:

- Either the real or the effective user ID of the sending process matches the real or effective user ID of the receiving process.
- The effective user ID of the sending process is superuser.

The processes that have the process IDs 0 and 1 are special processes and are sometimes referred to here as **proc0** and **proc1**, respectively.

If the *pid* parameter is greater than 0, the signal specified by the *sig* parameter is sent to the process whose process ID is equal to the value of the *pid* parameter.

If the *pid* parameter is equal to 0, the signal specified by the *sig* parameter is sent to all of the processes, excluding **proc0** and **proc1**, whose process group ID is equal to the process group ID of the sender.

If the *pid* parameter is equal to -1 and the effective user ID of the sender is not superuser, the signal specified by the *sig* parameter is sent to all of the processes, excluding **proc0** and **proc1**, whose real user ID is equal to the effective user ID of the sender.

If the *pid* parameter is equal to -1 and the effective user ID of the sender is superuser, the signal specified by the *sig* parameter is sent to all of the processes, excluding **proc0** and **proc1**.

If the *pid* parameter is negative but not -1, the signal specified by the *sig* parameter is sent to all of the processes whose process group ID is equal to the absolute value of the *pid* parameter.

Return Value

Upon successful completion, **kill** returns a value of 0. If **kill** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **kill** system call fails and no signal is sent if one or more of the following are true:

EINVAL *sig* is not a valid signal number.

EINVAL *sig* is **SIGKILL** and *pid* is 1 (**proc1**).

ESRCH No process can be found corresponding to that specified by *pid*.

EPERM The user ID of the sending process is not superuser, and the real or effective user ID does not match the real or effective user ID of the receiving process.

Related Information

In this book: “getpid, getpgrp, getppid” on page 2-52, “setpgrp” on page 2-122, and “signal” on page 2-139.

The **kill** command in *AIX Operating System Commands Reference*.

link

link

Purpose

Creates an additional directory entry for an existing file.

Syntax

```
int link (path1, path2)  
char *path1, *path2;
```

Description

The **link** system call creates an additional link (directory entry) for an existing file. The *path1* parameter points to the the path name of an existing file and the *path2* parameter points to the path name for the new directory entry to be created.

Return Value

Upon successful completion, **link** returns a value of 0. If **link** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **link** system call fails if one or more of the following are true:

- | | |
|----------------|--|
| ENOTDIR | A component of either path prefix is not a directory. |
| ENOENT | A component of either path prefix does not exist. |
| EACCESS | A component of either path prefix denies search permission. |
| ENOENT | The file named by the <i>path1</i> parameter does not exist. |
| EEXIST | The link named by the <i>path2</i> parameter already exists. |
| EPERM | The file named by the <i>path1</i> parameter is a directory and the effective user ID is not superuser. |
| EXDEV | The link named by the <i>path2</i> parameter and the file named by the <i>path1</i> parameter are on different file systems. |
| ENOENT | The <i>path2</i> parameter points to a null path name. |

EACCES	The requested link requires writing in a directory with a mode that denies write permission.
EROFS	The requested link requires writing in a directory on a read-only file system.
EFAULT	The <i>path1</i> or <i>path2</i> parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “unlink” on page 2-168.

The **link** command in *AIX Operating System Commands Reference*.

lockf

lockf

Purpose

Locks a region of a file for exclusive access.

Syntax

```
#include <sys/lockf.h >
```

```
int lockf (fildes, request, size)  
int fildes, request;  
off_t size;
```

Description

The **lockf** system call locks and unlocks regions of an open file. It is used to synchronize simultaneous access to a file by multiple processes. Only one process at a time can hold a lock on any given region of a file. Two types of locks are provided: enforced and advisory.

When a process holds an **enforced lock** on a region of a file, no other process can access that region with the **read** or **write** system calls. In addition, **creat** and **open** are prevented from truncating the file. If another process attempts to **read** or **write** the region, then it sleeps until the region is unlocked. However, if the system detects that sleeping would cause deadlock, then the **read** or **write** system call fails, setting **errno** to **EDEADLK**. If another process attempts to truncate the file with either the **creat** or **open** system call, then that system call fails and sets **errno** to **EACCES**.

When a process holds an **advisory lock** on a region of a file, no other process can lock that region, or an overlapping region, with **lockf**. The **read**, **write**, **creat**, and **open** system calls are not affected. This means that processes must voluntarily call **lockf** in order to make advisory locks effective.

Warning: Buffered I/O does not work properly when used with file locking. In particular, do not use the Standard I/O Package (**libc.a**) routines on files that are going to be locked, since these routines use buffered I/O.

To select enforced locking, the **S_ENFMT** bit must be set in the access permission code (or mode) of the file. Otherwise locking is advisory. Thus, a given file can have advisory or enforced locks, but not both.

The *filides* parameter to **lockf** is an open file descriptor obtained from a successful call to **open**, **creat**, **dup**, or **pipe** system call.

The *size* parameter is the number of bytes to be locked or unlocked. The region starts at the current location in the open file and extends forward if *size* is positive, or backward if *size* is negative. If the *size* parameter is 0, then the region starts at the current location and extends forward to the maximum possible file size, including the unallocated space after the end-of-file. Unallocated “holes” in the file can also be locked. (See “**fclear**” on page 2-40 about “holes” in files.)

The *request* parameter is one of the following constants:

- F_ULOCK** Unlocks a previously locked region in the file.
- F_LOCK** Locks the region for exclusive use. This *request* causes the calling process to sleep if the region overlaps a locked region, and to resume when it is granted the lock.
- F_TLOCK** Tests to see if another process has locked the specified region, and, if not, locks the region for exclusive use. If the region is already locked, then **lockf** fails and sets **errno** to **EACCES**.
- F_TEST** Tests to see if another process has already locked a region. **lockf** returns 0 if the region is unlocked. If the region is locked, then -1 is returned and **errno** is set to **EACCES**.

The system keeps a table of the locked regions for each file. This table can hold a limited number of entries. When the same process locks two regions that are next to each other in the file, **lockf** combines the lock table entries to conserve space in the lock table. An unlock request in the middle of a locked region leaves two locked regions, which can cause the lock table to overflow. When a lock or unlock request cannot be satisfied because the lock table is full, the **lockf** subroutine fails.

When a process closes a file, all of its locks on that file are removed. When a process terminates, all of the locks that it holds are removed.

All locks applied to directories, special files, and pipes are treated as advisory locks. However, locking directories is not recommended. Only advisory locks are supported for mapped files. An attempt to apply an enforced lock to a mapped file causes the **lockf** system call to fail and set **errno** to **EMFILE**. (For information about mapped files, see “**shmat**” on page 2-125.)

A child process does not inherit the locks of its parent process.

lockf

Return Value

Upon successful completion, **lockf** returns a value of 0. If **lockf** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **lockf** system call fails if one or more of the following are true:

- | | |
|----------------|--|
| EBADF | <i>filides</i> is not a valid open file descriptor. |
| EINVAL | <i>request</i> is not valid. |
| EACCES | F_TEST or F_TLOCK fails because another process has already locked the region. |
| EMFILE | The file is mapped and enforced locking is enabled. |
| EDEADLK | Deadlock will occur or the lock table is full. |

Related Information

In this book: “close” on page 2-25, “creat” on page 2-27, “dup” on page 2-30, “open” on page 2-88, “read, readx” on page 2-104, “write, writex” on page 2-178, and “stdio” on page 3-297.

lseek

Purpose

Moves read/write file pointer.

Syntax

```
#include <sys/types.h>
```

```
off_t lseek (fildes, offset, whence)  
int fildes;  
off_t offset;  
int whence;
```

Description

The **lseek** system call sets the file pointer for the file specified by the *fildes* parameter. The *fildes* parameter is a file descriptor obtained from a **creat**, **open**, **dup**, or **fcntl** system call.

The **lseek** system call sets the file pointer associated with the *fildes* stream as follows:

- If the *whence* parameter is 0, the pointer is set to the value of the *offset* parameter.
- If the *whence* parameter is 1, the pointer is set to its current location plus the value of the *offset* parameter.
- If the *whence* parameter is 2, the pointer is set to the size of the file plus the value of the *offset* parameter.

Return Value

Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned. If **lseek** fails, a value of -1 is returned and **errno** is set to indicate the error.

$(1024)_{10} \equiv 2^{10}$
 $\begin{array}{r} 1024 \\ 2 \overline{) 1024} \\ \underline{2048} \\ 2048 \\ \underline{4096} \\ 4096 \\ \underline{8192} \\ 8192 \\ \underline{16384} \\ 16384 \\ \underline{32768} \\ 32768 \\ \underline{65536} \\ 65536 \\ \underline{131072} \\ 131072 \\ \underline{262144} \\ 262144 \\ \underline{524288} \\ 524288 \\ \underline{1048576} \\ 1048576 \\ \underline{2097152} \\ 2097152 \\ \underline{4194304} \\ 4194304 \\ \underline{8388608} \\ 8388608 \\ \underline{16777216} \\ 16777216 \\ \underline{33554432} \\ 33554432 \\ \underline{67108864} \\ 67108864 \\ \underline{134217728} \\ 134217728 \\ \underline{268435456} \\ 268435456 \\ \underline{536870912} \\ 536870912 \\ \underline{1073741824} \\ 1073741824 \\ \underline{2147483648} \\ 2147483648 \\ \underline{4294967296} \\ 4294967296 \\ \underline{8589934592} \\ 8589934592 \\ \underline{17179869184} \\ 17179869184 \\ \underline{34359738368} \\ 34359738368 \\ \underline{68719476736} \\ 68719476736 \\ \underline{137438953472} \\ 137438953472 \\ \underline{274877906944} \\ 274877906944 \\ \underline{549755813888} \\ 549755813888 \\ \underline{1099511627776} \\ 1099511627776 \\ \underline{2199023255552} \\ 2199023255552 \\ \underline{4398046511104} \\ 4398046511104 \\ \underline{8796093022208} \\ 8796093022208 \\ \underline{17592186044416} \\ 17592186044416 \\ \underline{35184372088832} \\ 35184372088832 \\ \underline{70368744177664} \\ 70368744177664 \\ \underline{140737488355328} \\ 140737488355328 \\ \underline{281474976710656} \\ 281474976710656 \\ \underline{562949953421312} \\ 562949953421312 \\ \underline{1125899906842624} \\ 1125899906842624 \\ \underline{2251799813685248} \\ 2251799813685248 \\ \underline{4503599627370496} \\ 4503599627370496 \\ \underline{9007199254740992} \\ 9007199254740992 \\ \underline{18014398509481984} \\ 18014398509481984 \\ \underline{36028797018963968} \\ 36028797018963968 \\ \underline{72057594037927936} \\ 72057594037927936 \\ \underline{144115188075855872} \\ 144115188075855872 \\ \underline{288230376151711744} \\ 288230376151711744 \\ \underline{576460752303423488} \\ 576460752303423488 \\ \underline{1152921504606846976} \\ 1152921504606846976 \\ \underline{2305843009213693952} \\ 2305843009213693952 \\ \underline{4611686018427387904} \\ 4611686018427387904 \\ \underline{9223372036854775808} \\ 9223372036854775808 \\ \underline{18446744073709551616} \\ 18446744073709551616 \\ \underline{36893488147419103232} \\ 36893488147419103232 \\ \underline{73786976294838206464} \\ 73786976294838206464 \\ \underline{147573952589676412928} \\ 147573952589676412928 \\ \underline{295147905179352825856} \\ 295147905179352825856 \\ \underline{590295810358705651712} \\ 590295810358705651712 \\ \underline{1180591620717411303424} \\ 1180591620717411303424 \\ \underline{2361183241434822606848} \\ 2361183241434822606848 \\ \underline{4722366482869645213696} \\ 4722366482869645213696 \\ \underline{9444732965739290427392} \\ 9444732965739290427392 \\ \underline{18889465931478580854784} \\ 18889465931478580854784 \\ \underline{37778931862957161709568} \\ 37778931862957161709568 \\ \underline{75557863725914323419136} \\ 75557863725914323419136 \\ \underline{151115727451828646838272} \\ 151115727451828646838272 \\ \underline{302231454903657293676544} \\ 302231454903657293676544 \\ \underline{604462909807314587353088} \\ 604462909807314587353088 \\ \underline{1208925819614629174706176} \\ 1208925819614629174706176 \\ \underline{2417851639229258349412352} \\ 2417851639229258349412352 \\ \underline{4835703278458516698824704} \\ 4835703278458516698824704 \\ \underline{9671406556917033397649408} \\ 9671406556917033397649408 \\ \underline{19342813113834066795298816} \\ 19342813113834066795298816 \\ \underline{38685626227668133590597632} \\ 38685626227668133590597632 \\ \underline{77371252455336267181195264} \\ 77371252455336267181195264 \\ \underline{154742504910672534362390528} \\ 154742504910672534362390528 \\ \underline{309485009821345068724781056} \\ 309485009821345068724781056 \\ \underline{618970019642690137449562112} \\ 618970019642690137449562112 \\ \underline{1237940039285380274899124224} \\ 1237940039285380274899124224 \\ \underline{2475880078570760549798248448} \\ 2475880078570760549798248448 \\ \underline{4951760157141521099596496896} \\ 4951760157141521099596496896 \\ \underline{9903520314283042199192993792} \\ 9903520314283042199192993792 \\ \underline{19807040628566084398385987584} \\ 19807040628566084398385987584 \\ \underline{39614081257132168796771975168} \\ 39614081257132168796771975168 \\ \underline{79228162514264337593543950336} \\ 79228162514264337593543950336 \\ \underline{158456325028528675187087900672} \\ 158456325028528675187087900672 \\ \underline{316912650057057350374175801344} \\ 316912650057057350374175801344 \\ \underline{633825300114114700748351602688} \\ 633825300114114700748351602688 \\ \underline{1267650600228229401496703205376} \\ 1267650600228229401496703205376 \\ \underline{2535301200456458802993406410752} \\ 2535301200456458802993406410752 \\ \underline{5070602400912917605986812821504} \\ 5070602400912917605986812821504 \\ \underline{10141204801825835211973625643008} \\ 10141204801825835211973625643008 \\ \underline{20282409603651670423947251286016} \\ 20282409603651670423947251286016 \\ \underline{40564819207303340847894502572032} \\ 40564819207303340847894502572032 \\ \underline{81129638414606681695789005144064} \\ 81129638414606681695789005144064 \\ \underline{162259276829213363391578010288128} \\ 162259276829213363391578010288128 \\ \underline{324518553658426726783156020576256} \\ 324518553658426726783156020576256 \\ \underline{649037107316853453566312041152512} \\ 649037107316853453566312041152512 \\ \underline{1298074214633706907132624082305024} \\ 1298074214633706907132624082305024 \\ \underline{2596148429267413814265248164610048} \\ 2596148429267413814265248164610048 \\ \underline{5192296858534827628530496329220096} \\ 5192296858534827628530496329220096 \\ \underline{10384593717069655257060992658440192} \\ 10384593717069655257060992658440192 \\ \underline{20769187434139310514121985316880384} \\ 20769187434139310514121985316880384 \\ \underline{41538374868278621028243970633760768} \\ 41538374868278621028243970633760768 \\ \underline{83076749736557242056487941267521536} \\ 83076749736557242056487941267521536 \\ \underline{166153499473114484112975882535043072} \\ 166153499473114484112975882535043072 \\ \underline{332306998946228968225951765070086144} \\ 332306998946228968225951765070086144 \\ \underline{664613997892457936451903530140172288} \\ 664613997892457936451903530140172288 \\ \underline{1329227995784915872903807060280344576} \\ 1329227995784915872903807060280344576 \\ \underline{2658455991569831745807614120560689152} \\ 2658455991569831745807614120560689152 \\ \underline{5316911983139663491615228241121378304} \\ 5316911983139663491615228241121378304 \\ \underline{10633823966279326983230456482242756608} \\ 10633823966279326983230456482242756608 \\ \underline{21267647932558653966460912964485513216} \\ 21267647932558653966460912964485513216 \\ \underline{42535295865117307932921825928971026432} \\ 42535295865117307932921825928971026432 \\ \underline{85070591730234615865843651857942052864} \\ 85070591730234615865843651857942052864 \\ \underline{170141183460469231731687303715884105728} \\ 170141183460469231731687303715884105728 \\ \underline{340282366920938463463374607431768211456} \\ 340282366920938463463374607431768211456 \\ \underline{680564733841876926926749214863536422912} \\ 680564733841876926926749214863536422912 \\ \underline{1361129467683753853853498429727072845824} \\ 1361129467683753853853498429727072845824 \\ \underline{2722258935367507707706996859454145691648} \\ 2722258935367507707706996859454145691648 \\ \underline{5444517870735015415413993718908291383296} \\ 5444517870735015415413993718908291383296 \\ \underline{10889035741470030830827987437816582766592} \\ 10889035741470030830827987437816582766592 \\ \underline{21778071482940061661655974875633165533184} \\ 21778071482940061661655974875633165533184 \\ \underline{43556142965880123323311949751266331066368} \\ 43556142965880123323311949751266331066368 \\ \underline{87112285931760246646623899502532662132736} \\ 87112285931760246646623899502532662132736 \\ \underline{174224571863520493293247799005065324265472} \\ 174224571863520493293247799005065324265472 \\ \underline{348449143727040986586495598010130648530944} \\ 348449143727040986586495598010130648530944 \\ \underline{696898287454081973172991196020261297061888} \\ 696898287454081973172991196020261297061888 \\ \underline{1393796574908163946345982392040522594123776} \\ 1393796574908163946345982392040522594123776 \\ \underline{2787593149816327892691964784081045188247552} \\ 2787593149816327892691964784081045188247552 \\ \underline{5575186299632655785383929568162090376495104} \\ 5575186299632655785383929568162090376495104 \\ \underline{11150372599265311570767859136324180752990208} \\ 11150372599265311570767859136324180752990208 \\ \underline{22300745198530623141535718272648361505980416} \\ 22300745198530623141535718272648361505980416 \\ \underline{44601490397061246283071436545296723011960832} \\ 44601490397061246283071436545296723011960832 \\ \underline{89202980794122492566142873090593446023921664} \\ 89202980794122492566142873090593446023921664 \\ \underline{178405961588244985132285746181186892047843328} \\ 178405961588244985132285746181186892047843328 \\ \underline{356811923176489970264571492362373784095686656} \\ 356811923176489970264571492362373784095686656 \\ \underline{713623846352979940529142984724747568191373312} \\ 713623846352979940529142984724747568191373312 \\ \underline{1427247692705959881058285969449495136382746624} \\ 1427247692705959881058285969449495136382746624 \\ \underline{2854495385411919762116571938898990272765493248} \\ 2854495385411919762116571938898990272765493248 \\ \underline{5708990770823839524233143877797980545530986496} \\ 5708990770823839524233143877797980545530986496 \\ \underline{11417981541647679048466287755595961091061972992} \\ 11417981541647679048466287755595961091061972992 \\ \underline{22835963083295358096932575511191922182123945984} \\ 22835963083295358096932575511191922182123945984 \\ \underline{45671926166590716193865151022383844364247891968} \\ 45671926166590716193865151022383844364247891968 \\ \underline{91343852333181432387730302044767688728495783936} \\ 91343852333181432387730302044767688728495783936 \\ \underline{182687704666362864775460604089535377456991567872} \\ 182687704666362864775460604089535377456991567872 \\ \underline{365375409332725729550921208179070754913983135744} \\ 365375409332725729550921208179070754913983135744 \\ \underline{730750818665451459101842416358141509827966271488} \\ 730750818665451459101842416358141509827966271488 \\ \underline{1461501637330902918203684832716283019655932542976} \\ 1461501637330902918203684832716283019655932542976 \\ \underline{2923003274661805836407369665432566039311865085952} \\ 2923003274661805836407369665432566039311865085952 \\ \underline{5846006549323611672814739330865132078623730171904} \\ 5846006549323611672814739330865132078623730171904 \\ \underline{11692013098647223345629478661730264157247460343808} \\ 11692013098647223345629478661730264157247460343808 \\ \underline{23384026197294446691258957323460528314494920687616} \\ 23384026197294446691258957323460528314494920687616 \\ \underline{46768052394588893$

Diagnostics

The **lseek** system call fails and the file pointer remains unchanged if one or more of the following are true:

- EBADF** *fildev* is not an open file descriptor.
- ESPIPE** *fildev* is associated with a pipe or FIFO.
- EINVAL** *whence* is not 0, 1 or 2. This also causes a **SIGSYS** signal.
- EINVAL** The resulting file pointer would be negative.

Related Information

In this book: “creat” on page 2-27, “dup” on page 2-30, “fcntl” on page 2-42, “open” on page 2-88, and “fseek, rewind, ftell” on page 3-181.

mknod

Purpose

Creates a directory, or a special or ordinary file.

Syntax

```
#include <sys/stat.h>
```

```
int mknod (path, mode, dev)
char *path;
int mode;
dev_t dev;
```

Description

The **mknod** system call creates a new regular file, special file, or directory. The *path* parameter names the new file. The *mode* parameter specifies the *mode* of the file, which defines the file type and access permissions.

The *dev* parameter is configuration dependent and is used only if the *mode* parameter specifies a block or character special file. *dev* is the ID of the device, and it corresponds to the **st_rdev** member of the structure returned by the **stat** system call. See “stat, fstat” on page 2-153 and “stat.h” on page 5-61 for more information about the device ID.

The *mode* parameter is constructed logically OR-ing the values specified in “chmod” on page 2-18 with one the following values, which define the file type:

S_IFDIR	Directory
S_IFCHR	Character special file
S_IFMPX	Multiplexed character special file
S_IFBLK	Block special file
S_IFREG	Regular data file
S_IFIFO	FIFO special file

A complete list of the possible *mode* values and other useful macros appears in “stat.h” on page 5-61.

The low-order 9 bits of *mode* are modified by the process’s file mode creation mask: all bits set in the creation mask are cleared. (For more information about the creation mask, see “umask” on page 2-163.)

mknod

If the type of the new file is **S_IFMPX** (multiplexed character special file), then when the file is used, additional path name components can appear after the path name as if it were a directory. The additional part of the path name is available to the file's device driver for interpretation. This provides a multiplexed interface to the device driver. The **hft** device driver uses this feature. (See "hft" on page 6-19 for details about this device driver.)

The file's owner ID is set to the process's effective user ID. The file's group ID is set to the process's effective group ID.

The **mknod** system call can be invoked only by superuser for file types other than FIFO special.

Return Value

Upon successful completion, a value of 0 is returned. If the **mknod** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **mknod** system call fails and the new file is not created if one or more of the following are true:

EPERM	The process's effective user ID is not superuser.
ENOTDIR	A component of the path prefix is not a directory.
ENOENT	A component of the path prefix does not exist.
EROFS	The directory in which the file is to be created is located on a read-only file system.
EEXIST	The named file exists.
EFAULT	The <i>path</i> parameter points to a location outside of the process's allocated address space.

Related Information

In this book: "chmod" on page 2-18, "exec: execl, execv, execl, execve, execlp, execvp" on page 2-32, "umask" on page 2-163, "fs" on page 4-72, and "stat.h" on page 5-61.

The **chmod**, **mkdir**, and **mknod** commands in *AIX Operating System Commands Reference*.

mount

Purpose

Mounts a file system.

Syntax

```
int mount (dev, dir, rwflag)  
char *dev, *dir;  
int rwflag;
```

Description

The **mount** system call mounts a file system contained on the block device (also called a **special file**) identified by the *dev* parameter. The file system is mounted on the directory identified by the *dir* parameter. The **mount** system call can be used only by superuser.

The *dev* parameter and the *dir* parameter are pointers to path names.

The least significant bit of the *rwflag* parameter is used to control write permission on the mounted file system. If this bit is 1, writing is prohibited. If this bit is 0, writing is permitted according to individual file accessibility.

If the next least significant bit of *rwflag* is set, then *dev* is mounted as a **removable** file system. Whenever there are no active references to files or directories on the file system, the operating system forgets the content and structure of the file system. The user can remove the medium and replace it with a different file system. All future references to *dir* will refer to the file system on the new medium.

After the file system is mounted, references to the path name specified by the *dir* parameter refer to the root directory on the mounted file system.

Return Value

Upon successful completion a value of 0 is returned. If the **mount** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **mount** system call fails if one or more of the following are true:

EPERM	The effective user ID of the calling process is not superuser.
ENOENT	<i>dev</i> or <i>dir</i> does not exist.
ENOTDIR	A component of a path prefix is not a directory.
ENOTBLK	<i>dev</i> is not a block device.
ENXIO	The device or driver for <i>dev</i> is not currently configured.
ENOTDIR	<i>dir</i> is not a directory.
EFAULT	The <i>dev</i> or <i>dir</i> parameter points to a location outside of the process's allocated address space.
EBUSY	<i>dir</i> is currently busy. For example, it may some process's current directory, or a file system may be mounted onto it.
EBUSY	The device associated with <i>dev</i> is currently mounted.
EBUSY	There are no more mount table entries.
EINVAL	The data on <i>dev</i> is not recognizable as a file system. This usually means that it does not contain a properly formatted super-block.

Related Information

In this book: “umount” on page 2-164, “fs” on page 4-72.

The **mount** and **umount** commands in *AIX Operating System Commands Reference*.

msgctl

Purpose

Provides message control operations.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

Description

The **msgctl** system call provides a variety of message control operations as specified by *cmd* parameter. The following *cmds* are available:

IPC_STAT Stores the current value of each member of the data structure associated with the *msqid* parameter into the structure pointed to by the *buf* parameter. The current process must have read permission in order to perform this operation. The **msqid_ds** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
struct ipc_perm msg_perm; /* Operation permission structure */
struct msg *msg_first; /* Ptr to first message on the queue */
struct msg *msg_last; /* Ptr to last message on the queue */
ushort msg_cbytes; /* Current number of bytes on the queue */
ushort msg_qnum; /* Number of messages on the queue */
ushort msg_qbytes; /* Maximum number of bytes on the queue */
ushort msg_lspid; /* ID of last process to call msgsnd */
ushort msg_lrpid; /* ID of last process to call msgrcv */
time_t msg_stime; /* Time of last msgsnd call */
time_t msg_rtime; /* Time of last msgrcv call */
time_t msg_ctime; /* Time of the last change to this */
/* structure with a msgctl call */
```


msgctl

IPC_SET Sets the value of the following members of the data structure associated with the *msqid* parameter to the corresponding values found in the structure pointed to by the *buf* parameter:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* Only the low-order nine bits */
msg_qbytes
```

The current process must have an effective user ID equal to either that of superuser or to the value of **msg_perm.uid** in the data structure associated with *msqid* in order to perform this operation. To raise the value of **msg_qbytes**, the effective user ID of the current process must be superuser.

IPC_RMID Removes the message queue identifier specified by the *msqid* parameter from the system and destroys the message queue and data structure associated with it. The current process must have an effective user ID equal to either that of superuser or to the value of **msg_perm.uid** in the data structure associated with *msqid* in order to perform this operation.

Return Value

Upon successful completion, a value of 0 is returned. If the **msgctl** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **msgctl** system call fails if one or more of the following are true:

EINVAL *msqid* is not a valid message queue identifier.

EINVAL *cmd* is not a valid command.

EACCES *cmd* is equal to **IPC_STAT** and read permission is denied to the calling process.

EPERM *cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of superuser and it is not equal to the value of **msg_perm.uid** in the data structure associated with *msqid*.

EPERM *cmd* is equal to **IPC_SET**, an attempt is being made to increase to the value of **msg_qbytes**, and the effective user ID of the calling process is not equal to that of superuser.

EFAULT The *buf* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “msgget” on page 2-74, “msgsnd” on page 2-80, “msgrcv” on page 2-77, and “msgxrcv” on page 2-83.

msgget

msgget

Purpose

Gets a message queue identifier.

Syntax

```
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int msgget (key, msgflg)
key_t key;
int msgflg;
```

Description

The **msgget** system call returns the message queue identifier associated with the specified *key*. The *key* parameter is either the value **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm). See “ftok” on page 3-183 for details about this subroutine.

The *msgflg* parameter is constructed by logically OR-ing one or more of the following values:

IPC_CREAT	Creates the data structure if it does not already exist.
IPC_EXCL	Causes the msgget system call to fail if IPC_CREAT is also set and the data structure already exists.
S_IRUSR	Permits the process that owns the data structure to read it.
S_IWUSR	Permits the process that owns the data structure to modify it.
S_IRGRP	Permits the group associated with the data structure to read it.
S_IWGRP	Permits the group associated with the data structure to modify it.
S_IROTH	Permits others to read the data structure.
S-IWOTH	Permits others to modify the data structure.

The values that begin with **S-I** are defined in the **sys/stat.h** header file and are a subset of the access permissions that apply to files.

A message queue identifier and associated message queue and data structure are created for the value of the *key* parameter if one of the following are true:

- *key* is equal to **IPC_PRIVATE**.
- *key* does not already have a message queue identifier associated with it, and **IPC_CREAT** is set.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- **msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order nine bits of **msg_perm.mode** are set equal to the low-order nine bits of the *msgflg* parameter.
- **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set equal to 0.
- **msg_ctime** is set equal to the current time.
- **msg_qbytes** is set equal to the system limit.

Return Value

Upon successful completion, a message queue identifier is returned. If the **msgget** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **msgget** system call fails if one or more of the following are true:

EACCES	A message queue identifier exists for the <i>key</i> parameter but operation permission as specified by the low-order nine bits of the <i>msgflg</i> parameter would not be granted.
ENOENT	A message queue identifier does not exist for the <i>key</i> parameter and IPC_CREAT is not set.
ENOSPC	A message queue identifier is to be created but the system imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.
EEXIST	A message queue identifier exists for <i>key</i> , and both IPC_CREAT and IPC_EXCL are set.

Related Information

In this book: “msgctl” on page 2-71, “msgsnd” on page 2-80, “msgrcv” on page 2-77, “msgxrcv” on page 2-83, and “ftok” on page 3-183.

msgrcv

Purpose

Reads a message from a queue.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Description

The **msgrcv** system call reads a message from the queue specified by the *msqid* parameter and stores it into the structure pointed to by the *msgp* parameter. The current process must have read permission in order to perform this operation. The **msgbuf** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
long  mtype;      /* Message type */
char  mtext[1];   /* Beginning of message text */
```

The **mtype** field contains the type of the received message as specified by the sending process. **mtext** is the text of the message.

The *msgsz* parameter specifies the size of **mtext** in bytes. The received message is truncated to the size specified by the *msgsz* parameter if it is longer than the size specified by the *msgsz* parameter and if **MSG_NOERROR** is set in *msgflg*. The truncated part of the message is lost and no indication of the truncation is given to the calling process. If the message is longer than *msgsz* bytes and **MSG_NOERROR** is not set, then the **msgrcv** system call fails and sets **errno** to **E2BIG**.

The *msgtyp* parameter specifies the type of message requested as follows:

- If the *msgtyp* parameter is equal to 0, the first message on the queue is received.

msgrcv

- If the *msgtyp* parameter is greater than 0, the first message of the type specified by the *msgtyp* parameter is received.
- If the *msgtyp* parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the *msgtyp* parameter is received.

The *msgflg* parameter is either 0, or is constructed by logically OR-ing one or more of the following values:

MSG_NOERROR Truncates the message if it is longer than *msgsz* bytes.

IPC_NOWAIT Specifies the action to take if a message of the desired type is not on the queue:

- If **IPC_NOWAIT** is set, then the calling process returns a value of -1 and sets **errno** to **ENOMSG**.
- If **IPC_NOWAIT** is not set, then the calling process suspends execution until one of the following occurs:
 - A message of the desired type is placed on the queue.
 - The message queue identifier specified by the *msqid* parameter is removed from the system. When this occurs, **errno** is set to **EIDRM**, and a value of -1 is returned.
 - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner described in “signal” on page 2-139.

Return Value

Upon successful completion, **msgrcv** returns a value equal to the number of bytes actually stored into **mtext** and the following actions are taken with respect to the data structure associated with the *msqid* parameter:

- **msg_qnum** is decremented by 1.
- **msg_lrpid** is set equal to the process ID of the calling process.
- **msg_rtime** is set equal to the current time.

If the **msgrcv** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **msgrcv** system call fails if one or more of the following are true:

EINVAL	<i>msqid</i> is not a valid message queue identifier.
EACCES	Operation permission is denied to the calling process.
EINVAL	<i>msgsz</i> is less than 0.
E2BIG	<i>mtext</i> is greater than <i>msgsz</i> and MSG_NOERROR is not set.
ENOMSG	The queue does not contain a message of the desired type and IPC_NOWAIT is set.
EFAULT	The <i>msgp</i> parameter points to a location outside of the process's allocated address space.
EINTR	msgrcv received a signal.
EIDRM	The message queue identifier specified by <i>msqid</i> has been removed from the system.

Related Information

In this book: “msgctl” on page 2-71, “msgget” on page 2-74, “msgsnd” on page 2-80, “msgxrcv” on page 2-83, and “signal” on page 2-139.

msgsnd

msgsnd

Purpose

Sends a message.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;
```

Description

The **msgsnd** system call sends a message to the queue specified by the *msqid* parameter. The current process must have write permission in order to perform this operation. The *msgp* parameter points to a **msgbuf** structure containing the message. The **msgbuf** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
long  mtype;      /* Message type */
char  mtext[1];   /* Beginning of message text */
```

The *mtype* parameter is a positive integer that is used by the receiving process for message selection. The *mtext* parameter is any text of the length in bytes specified by the *msgsz* parameter. The *msgsz* parameter can range from 0 to a system-imposed maximum.

The *msgflg* parameter specifies the action to be taken if the message cannot be sent for one of the following reasons:

- The number of bytes already on the queue is equal to **msg_qbytes**.
- The total number of messages on all queues system-wide is equal to a system-imposed limit.

These actions are as follows:

- If *msgflg* is set to **IPC_NOWAIT**, then the message is not sent, and **msgsnd** returns a value of -1 and sets **errno** to **EAGAIN**.
- If *msgflg* is 0, then the calling process suspends execution until one of the following occurs:
 - The condition responsible for the suspension no longer exists, in which case the message is sent.
 - *msqid* is removed from the system. (For information on how to remove *msqid*, see “msgctl” on page 2-71.) When this occurs, **errno** is set equal to **EIDRM**, and a value of -1 is returned.
 - The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in “signal” on page 2-139.

Return Value

Upon successful completion, a value of 0 is returned and the following actions are taken with respect to the data structure associated with the *msqid* parameter:

- **msg_qnum** is incremented by 1.
- **msg_lspid** is set equal to the process ID of the calling process.
- **msg_stime** is set equal to the current time.

If the **msgsnd** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **msgsnd** system call fails and no message is sent if one or more of the following are true:

EINVAL	The <i>msqid</i> parameter is not a valid message queue identifier.
EACCES	Operation permission is denied to the calling process.
EINVAL	mtype is less than 1.
EAGAIN	The message cannot be sent for one of the reasons stated previously, and <i>msgflg</i> is set to IPC_NOWAIT .
EINVAL	The <i>msgsz</i> parameter is less than 0 or greater than the system-imposed limit.
EFAULT	The <i>msgp</i> parameter points to a location outside of the process's allocated address space.

msgsnd

EINTR	msgsnd received a signal.
EIDRM	The message queue identifier specified by <i>msqid</i> has been removed from the system.

Related Information

In this book: “msgctl” on page 2-71, “msgget” on page 2-74, “msgrcv” on page 2-77, “msgxrcv” on page 2-83, and “signal” on page 2-139.

msgxrcv

Purpose

Receives an extended message.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgxrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgxbuf *msgp;
int msgsz, msgflg;
long msgtyp;
```

Description

The **msgxrcv** system call reads a message from the queue specified by the *msqid* parameter and stores it into the extended message receive buffer pointed to by the *msgp* parameter. The current process must have read permission in order to perform this operation. The **msgxbuf** structure is defined in the **sys/msg.h** header file, and it contains the following members:

time_t	mtime;	/* Time message was sent */
short	muid;	/* Sender's effective user ID */
short	mgid;	/* Sender's effective group ID */
long	mnid;	/* Sender's node ID */
short	mpid;	/* Sender's process ID */
long	mtype;	/* Message type */
char	mtext[1];	/* Beginning of message text */

The *msgsz* parameter specifies the size of **mtext** in bytes. The receive message is truncated to the size specified by the *msgsz* parameter if it is larger than the *msgsz* parameter and **MSG_NOERROR** is true. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

The *msgsz* parameter specifies the size of **mtext** in bytes. The received message is truncated to the size specified by the *msgsz* parameter if it is larger than the size specified by the *msgsz* parameter and if **MSG_NOERROR** is set in *msgflg*. The truncated part of

the message is lost and no indication of the truncation is given to the calling process. If the message is longer than *msgsz* bytes and **MSG_NOERROR** is not set, then the **msgrcv** system call fails and sets **errno** to **E2BIG**.

The *msgtyp* parameter specifies the type of message requested as follows:

- If the *msgtyp* parameter is equal to 0, the first message on the queue is received.
- If the *msgtyp* parameter is greater than 0, the first message of the type specified by the *msgtyp* parameter is received.
- If the *msgtyp* parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the *msgtyp* parameter is received.

The *msgflg* parameter is either 0, or is constructed by logically OR-ing one or more of the following values:

MSG_NOERROR Truncates the message if it is longer than *msgsz* bytes.

IPC_NOWAIT Specifies the action to take if a message of the desired type is not on the queue:

- If **IPC_NOWAIT** is set, then the calling process returns a value of -1 and sets **errno** to **ENOMSG**.
- If **IPC_NOWAIT** is not set, then the calling process suspends execution until one of the following occurs:
 - A message of the desired type is placed on the queue.
 - The message queue identifier specified by the *msqid* parameter is removed from the system. When this occurs, **errno** is set to **EIDRM**, and a value of -1 is returned.
 - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner prescribed in “signal” on page 2-139.

Return Value

Upon successful completion, **msgxrcv** returns a value equal to the number of bytes actually stored into *mtext*, and the following actions are taken with respect to the data structure associated with the *msqid* parameter:

- **msg_qnum** is decremented by 1.
- **msg_lrpuid** is set equal to the process ID of the calling process.
- **msg_rtime** is set equal to the current time.

If the **msgxrcv** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **msgxrcv** system call fails if one or more of the following are true:

- | | |
|---------------|--|
| EINVAL | <i>msqid</i> is not a valid message queue identifier. |
| EACCES | Operation permission is denied to the calling process. |
| EINVAL | <i>msgsz</i> is less than 0. |
| E2BIG | <i>mtext</i> is greater than <i>msgsz</i> and MSG_NOERROR is not set. |
| ENOMSG | The queue does not contain a message of the desired type and IPC_NOWAIT is set. |
| EFAULT | The <i>msgp</i> parameter points to a location outside of the process's allocated address space. |
| EINTR | msgxrcv received a signal. |
| EIDRM | The message queue identifier specified by <i>msqid</i> is removed from the system. |

Related Information

In this book: “msgctl” on page 2-71, “msgget” on page 2-74, and “msgrcv” on page 2-77.

nice

nice

Purpose

Changes the priority of a process.

Syntax

```
int nice (incr)  
int incr;
```

Description

The **nice** system call adds the value of the *incr* parameter to the nice value of the calling process. A process's **nice value** is a positive number that determines that process's CPU priority. A higher number results in a lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. If *incr* causes the nice value to fall outside this range, then **nice** sets the nice value to the corresponding limit.

Return Value

Upon successful completion, the new nice value minus 20 is returned. If **nice** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **nice** system call fails and the nice value is not changed if:

EPERM	The <i>incr</i> parameter is negative or the resulting nice value would be greater than 40, and the effective user ID of the calling process is not superuser.
--------------	--

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32.

The **nice** command in *AIX Operating System Commands Reference*.

open

open

Purpose

Opens a file for reading or writing.

Syntax

```
#include <fcntl.h>
```

```
int open (path, oflag [, mode])  
char *path;  
int oflag, mode;
```

Description

The **open** system call opens a file descriptor for the file named by the *path* parameter. The file status flags are set according to the value of the *oflag* parameter. The *oflag* parameter values are constructed by logically OR-ing flags from the following list:

Note: Do not use **O_RDONLY**, **O_WRONLY**, or **O_RDWR** together.

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY Open with no delay. This flag may affect subsequent reads and writes.

When opening a FIFO with **O_RDONLY** or **O_WRONLY** set:

- If **O_NDELAY** is set, an **open** for reading-only returns without delay. An **open** for writing-only returns an error if no process currently has the file open for reading.
- If **O_NDELAY** is clear, an **open** for reading-only blocks until a process opens the file for writing. An **open** for writing-only blocks until a process opens the file for reading.

When opening a file associated with a communication line:

- If **O_NDELAY** is set, the **open** returns without waiting for carrier.
- If **O_NDELAY** is clear, the **open** blocks until carrier is present.

When opening a regular file that supports enforced record locks:

- If **O_NDELAY** is set, then reads and writes to portions of the file that are locked by other processes return an error.
- If **O_NDELAY** is clear, then reads and writes to portions of the file that are locked by other processes blocks until the locks are released.

O_APPEND If set, the file pointer is set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. If the file does not exist, then the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of the *mode* parameter modified as follows:

- All bits set in the process's file mode creation mask are cleared. (For information about the creation mask, see "umask" on page 2-163.)
- The **S_ISVTX** bit of the mode, which saves the text image after execution, is cleared.

For information about file modes and a list of the mode values, see "chmod" on page 2-18 and "stat.h" on page 5-61.)

O_TRUNC If the file exists, then its length is truncated to 0, and the mode and owner are unchanged. If the file has any outstanding record locks, then **open** fails and the file remains unchanged.

O_EXCL If **O_EXCL** and **O_CREAT** are set, **open** fails if the file exists.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across **exec** system calls (see "fcntl" on page 2-42).

No process can have more than 200 file descriptors open simultaneously.

Return Value

Upon successful completion, the file descriptor, a nonnegative integer, is returned. If **open** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **open** system call fails, and the named file is not opened if one or more of the following are true:

ENOTDIR	A component of the path prefix is not a directory.
ENOENT	O_CREAT is not set and the named file does not exist.
EACCES	A component of the path prefix denies search permission.
EACCES	The type of access specified by the <i>oflag</i> parameter is denied for the named file.
EISDIR	The named file is a directory and the <i>oflag</i> parameter is write or read/write.
EROFS	The named file resides on a read-only file system and the <i>oflag</i> parameter is write or read/write.
EMFILE	Two hundred (200) file descriptors are currently open.
ENXIO	The named file is a character special or block special file, and the device associated with this special file does not exist.
ENXIO	The named file is a multiplexed special file and either the channel number is outside of the valid range, or no more channels are available.
ETXTBSY	The file is a pure procedure (shared text) file that is being executed and the <i>oflag</i> parameter is write or read/write.
EFAULT	The <i>path</i> parameter points to a location outside of the process's allocated address space.
EEXIST	O_CREAT and O_EXCL are set, and the named file exists.
ENXIO	O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.
EAGAIN	O_TRUNC is set, and the named file contains a record lock owned by another process. See "lockf" on page 2-62 for information about record locks.
EINTR	A signal was caught during the open system call.
ENFILE	The system file table is full.
ENOSPC	The directory that would contain the new file cannot be extended.

Related Information

In this book: “chmod” on page 2-18, “close” on page 2-25, “creat” on page 2-27, “dup” on page 2-30, “fcntl” on page 2-42, “lockf” on page 2-62, “lseek” on page 2-65, “mknod” on page 2-67, “read, readx” on page 2-104, “umask” on page 2-163, “write, writex” on page 2-178, and “stat.h” on page 5-61.

pause

pause

Purpose

Suspends a process until a signal is received.

Syntax

```
int pause ( )
```

Description

The **pause** system call suspends the calling process until it receives a signal. The signal must not be one that is ignored by the calling process. **pause** does not affect the action taken upon the receipt of a signal.

If the signal received causes the calling process to terminate, then the **pause** system call does not return.

If the signal is caught by the calling process and control is returned from the signal-catching function, then the calling process resumes execution from the point of suspension; the **pause** system call returns a value of -1 and sets **errno** to **EINTR**. (For information about signal-catching functions, see “signal” on page 2-139.)

Related Information

In this book: “alarm” on page 2-13, “kill” on page 2-58, “signal” on page 2-139, and “wait” on page 2-176.

pipe

Purpose

Creates an interprocess channel.

Syntax

```
int pipe (fildes)  
int fildes[2];
```

Description

The **pipe** system call creates an interprocess channel called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. The *fildes*[0] file descriptor is opened for reading and *fildes*[1] is opened for writing.

A read on file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out basis.

When writing, at least 5,120 bytes of data are buffered by the pipe before the writing process is blocked.

Warning: The actions of the **pipe** system call are undefined if the *fildes* parameter points to a location outside of the process's allocated address space.

Return Value

Upon successful completion, a value of 0 is returned. If **pipe** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **pipe** system call fails if one or more the following are true:

- | | |
|---------------|--|
| EMFILE | 199 or more file descriptors are already open. |
| ENFILE | The system file table is full. |

Related Information

In this book: “read, readx” on page 2-104 and “write, writex” on page 2-178.

The **sh** command in *AIX Operating System Commands Reference*.

plock

Purpose

Locks the process, text, or data in memory.

Syntax

```
#include <sys/lock.h>
```

```
int plock (op)  
int op;
```

Description

The **plock** system call allows the calling process to lock or unlock its text segment (*text lock*), its data segment (*data lock*), or both its text and data segments (*process lock*) into memory. Locked segments are pinned in memory and are immune to all routine paging. The effective user ID of the calling process must be superuser to use this call.

The *op* parameter specifies the following:

PROCLOCK	Locks text and data segments into memory (process lock).
TEXTLOCK	Locks text segment into memory (text lock).
DATLOCK	Locks data segment into memory (data lock).
UNLOCK	Removes locks.

Return Value

Upon successful completion, a value of 0 is returned to the calling process. If **plock** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **plock** system call fails if one or more of the following are true:

EPERM	The effective user ID of the calling process is not superuser.
EINVAL	The <i>op</i> parameter has a value other than PROCLOCK , TEXTLOCK , DATLOCK , or UNLOCK .

plock

- EINVAL** *op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.
- EINVAL** *op* is equal to **TXTLOCK** and a text lock, or a process lock already exists on the calling process.
- EINVAL** *op* is equal to **DATLOCK** and data lock, or a process lock already exists on the calling process.
- EINVAL** *op* is equal to **UNLOCK** and no type of lock exists on the calling process.

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “exit, _exit” on page 2-38, and “fork” on page 2-44.

profil

Purpose

Starts and stops execution profiling.

Syntax

```
#include <mon.h>

void profil (shortbuff, bufsiz, offset, scale)
    — or —
void profil (profbuff, -1, 0, 0)

short *shortbuff;
struct prof *profbuff;
unsigned int bufsiz, offset, scale;
```

Description

The **profil** system call arranges to record a histogram of periodically sampled values of the calling process's program counter.

If the *bufsiz* parameter has any value but -1, then the parameters to **profil** are interpreted as shown in the first syntax definition. The *shortbuff* parameter points to an area of memory, and its length (in bytes) is given by the *bufsiz* parameter.

After this call, the user's program counter (pc) is examined 60 times a second. The value of the *offset* parameter is subtracted from the pc, and the result is multiplied by the value of the *scale* parameter. If the resulting number is less than $\text{bufsiz} \div \text{sizeof}(\text{short})$, then the corresponding **short** inside *shortbuff* is incremented.

The least significant 16 bits of the *scale* parameter are interpreted as an unsigned, fixed-point fraction with a binary point at the left. The most significant 16 bits of *scale* are ignored. For example:

profil

Octal	Hex	Meaning
0177777	0xFFFF	Maps approximately each pair of bytes in the instruction space to a unique short in <i>shortbuff</i> .
077777	0x7FFF	Maps approximately every four bytes to a short in <i>shortbuff</i> .
01	0x0001	Maps all instructions to the first short in <i>shortbuff</i> , producing a noninterrupting core clock.
0	0x0000	Turns profiling off.

Mapping each byte of the instruction space to an individual **short** in *shortbuff* is not possible.

If the second parameter (*bufsize*) has the value -1, then the parameters to **profil** are interpreted as shown in the second syntax definition. In this case, the *offset* and *scale* parameters are ignored, and *profbuff* points to an array of **prof** structures. The **prof** structure is defined in the **mon.h** header file, and it contains the following members:

```
daddr_t  p_low;
daddr_t  p_high;
short_t   *p_buff;
int_t     p_bufsize;
int_t     p_scale;
```

If the **p-scale** member has the value -1, then a value for it is computed based on **p-low**, **p-high**, and **p-bufsize**; otherwise **p-scale** is interpreted like the **scale** argument in the first synopsis. The **p-high** members in successive structures must be in ascending sequence. The array of structures is terminated with a structure containing a **p-high** member set to zero.

Profiling is turned off:

- If the value of the *scale* parameter is 0.
- When an **exec** system call is executed
- If updating the buffer pointed to by the *shortbuff* or *profbuff* parameter would cause a memory fault.

Profiling is rendered ineffective by giving a value of 0 for the *bufsiz* parameter.

Profiling remains on in both the child process and the parent process after a **fork** system call.

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “fork” on page 2-44, and “monitor” on page 3-236.

The **cc** and **prof** commands in *AIX Operating System Commands Reference*.

ptrace

ptrace

Purpose

Traces the execution of a child process.

Syntax

```
int ptrace (request, pid, addr, data, buff)
int request, pid, addr, data, buff;
```

Description

The **ptrace** system call allows a parent process to control the execution of a child process. **ptrace** is primarily used by utility programs to implement breakpoint debugging. The **sdb** command described in *AIX Operating System Commands Reference* is such a debugging utility.

The child process behaves normally until it encounters a signal, at which time it enters a stopped state and its parent process is notified with the **wait** system call. When the child process is in the stopped state, its parent process can examine and modify its memory image using the **ptrace** system call. Also, the parent process can cause the child process to either terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* parameter determines the action to be taken by the **ptrace** system call and is one of the following:

- 0 This request must be issued by the child process that is to be traced by its parent. This request sets the child's trace flag that causes the child to be left in a stopped state upon receipt of a signal, rather than the state specified by the *func* parameter of the **signal** system call. The *pid*, *addr*, and *data* parameters are ignored, and a return value is not defined for this request. Do not issue this request if the parent does not expect to trace the child.

Note: The remainder of the requests can only be used by the parent process. For each request, the *pid* parameter is the process ID of the child. The child must be in a stopped state before these requests are made.

- 1, 2 These requests return the **int** in the child's address space at the location pointed to by the *addr* parameter. Either request 1 or request 2 can be used with with equal results. The *data* parameter is ignored. These requests fail if the value of the *addr*

parameter is not in the address space of the child process, in which case a value of -1 is returned, and the parent's **errno** is set to **EIO**.

- 3 This request returns the user area of the system's address space at the location pointed to by the *addr* parameter. (For information about the child's user area, see the **sys/user.h** header file.) The *data* parameter is ignored. The value of the *addr* parameter is rounded down to the the next **int** (word) boundary. This request fails if the *addr* parameter is outside the user area, in which case a value of -1 is returned to the parent process and the parent process's **errno** is set to **EIO**.
- 4, 5 These requests write the value of the *data* parameter into the address space of the child process at the **int** pointed to by the *addr* parameter. Either request 4 or request 5 can be used with equal results. Upon successful completion, the value written into the address space of the child process is returned to the parent process. These requests fail if the *addr* parameter points to a location in a pure procedure space and another process is executing in that space. They also fail if the *addr* is out of range. Upon failure, a value of -1 is returned to the parent process and the parent process's **errno** is set to **EIO**.
- 6 This request writes the value of the *data* parameter into the child's user area of the system's address space at the **int** pointed to by the *addr* parameter. The value of the *addr* parameter is rounded down to the the next **int** (word) boundary. Only the following entries can be modified:

IAR	Instruction Address Register
MQR	Multiply/Quotient Register
CS	Condition Status Register
- 7 This request causes the child process to resume execution. If the *data* parameter is 0, all pending signals, including the one that caused the child process to stop, are canceled before the child process resumes execution. If the *data* parameter is a valid signal number, the child process resumes execution as if it had received that signal. Any other pending signals are canceled. The *addr* parameter must be equal to 1 for this request. Upon successful completion, the value of the *data* parameter is returned to the parent process. This request fails if the *data* parameter is not 0 or a valid signal number, in which case a value of -1 is returned to the parent process and the parent process's **errno** is set to **EIO**.
- 8 This request causes the child process to terminate the same way it would with an **exit** system call.
- 11 This request returns the contents of one of the general-purpose registers of the child process. The *addr* parameter specifies which of the sixteen 32-bit registers is to be returned. The *data* and *buff* parameters are ignored. This request fails if the value of the *addr* parameter is not between 0 and 15 inclusive. In this case, **ptrace** returns the value -1 and sets the parent's **errno** to **EIO**.

- 12 This request returns the value of the floating-point register specified by the *addr* parameter. The *addr* parameter must be an value in the range from 0 to 7, excluding 6. Note that the status register (register 7) is only four bytes long. The *data* and *buff* parameters are ignored.
- 14 This request stores the value of the *data* parameter in one of the child process's general-purpose registers. The *addr* parameter specifies the register to be modified. The *buff* parameter is ignored. Upon successful completion, the value of *data* is returned to the parent process. This request fails if the value of the *addr* parameter is not between 0 and 15 inclusive. In this case, **ptrace** returns the value -1 and sets the parent's **errno** to **EIO**.
- 15 This request sets the floating-point register specified by the *addr* parameter. to the value given in the *data* parameter. The *addr* parameter must be an value in the range from 0 to 7, excluding 6. Note that the status register (register 7) is only four bytes long. The *data* and *buff* parameters are ignored.
- 17 This request reads a block of data from the child process's address space. The *addr* parameter points to the block of data in the child's address space and the *data* parameter gives its length in bytes. The value of the *data* parameter must not be greater than 1024. The *buff* parameter points to the location in the parent's address space into which the data is to be copied. Upon successful completion, **ptrace** returns the value of the *data* parameter. If an error occurs, **ptrace** returns -1 and sets the parent's **errno** to indicate the error. This request fails when one or more of the following are true:
- EINVAL** The *data* parameter is less than 1 or greater than 1024.
 - EIO** The *addr* parameter is not a valid pointer into the child process's address space.
 - EFAULT** The *buff* parameter does not point to a writable location in the parent process's address space.
- 19 This request writes a block of data into the child process's address space. The *addr* parameter points to the location in the child's address space to be written into. The *data* parameter gives the length of the block in bytes, and it must not be greater than 1024. The *buff* parameter points to the data in the parent's address space to be copied. Upon successful completion, the value of *data* is returned to the parent. If an error occurs, **ptrace** returns -1 and sets the parent's **errno** to indicate the error. This request fails when one or more of the following are true:
- EINVAL** The *data* parameter is less than 1 or greater than 1024.
 - EIO** The *addr* parameter is not a valid pointer into the child process's address space.
 - EFAULT** The *buff* parameter does not point to a readable location in the parent process's address space.

As a security measure, the **ptrace** system call inhibits the set-user-ID facility on subsequent **exec** system calls.

If a traced process initiates an **exec** system call, it stops before executing the first instruction of the new image and shows the signal **SIGTRAP**.

Diagnostics

In general, the **ptrace** system call fails if one or more of the following are true:

- | | |
|--------------|---|
| EIO | The <i>request</i> parameter is not one of the values listed. |
| ESRCH | The <i>pid</i> parameter identifies a child process that does not exist or has not executed a ptrace system call with request 0. |

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “signal” on page 2-139, and “wait” on page 2-176.

The **sdb** command in *AIX Operating System Commands Reference*.

read, readx

read, readx

Purpose

Reads from file.

Syntax

```
int read (fildes, buf, nbyte)  
int fildes;  
char *buf;  
unsigned int nbyte;
```

```
int readx (fildes, buf, nbyte, ext)  
int fildes, ext;  
char *buf;  
unsigned int nbyte;
```

Description

The **read** system call reads a set number of bytes into a buffer. The **read** system call reads the number of bytes set by the *nbyte* parameter from the file associated with the *fildes* parameter and places those bytes into the buffer pointed to by the *buf* parameter.

The *fildes* parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call.

On devices capable of seeking, the **read** starts at a position in the file given by the file pointer associated with the *fildes* parameter. Upon return from the **read** system call, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

When attempting to read from an empty pipe (or FIFO):

- If **O_NDELAY** is set, the **read** returns 0.
- If **O_NDELAY** is clear, the **read** blocks until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a terminal that has no data currently available:

- If **O_NDELAY** is set, the **read** returns 0.
- If **O_NDELAY** is clear, the **read** blocks until data becomes available.

When attempting to read a regular file that supports enforcement mode record locks, and all or part of the region to be read is currently locked by another process:

- If **O_NDELAY** is set, then the **read** returns -1 and sets **errno** to **EAGAIN**.
- If **O_NDELAY** is clear, then the **read** blocks the calling process until the lock is released.

For more information about record locks, see “lockf” on page 2-62.

If the file has been mapped, the **read** system call reads from a mapped file segment. If the *fildes* file descriptor was used to map the file copy-on-write, then the copy-on-write segment is used. Otherwise, the **read** system call reads from the read-write mapped segment for the file. See “shmat” on page 2-125 for information about mapping files.

The **readx** system call performs the same function as **read**, except that it provides communication with character device drivers that require more information or return more status than **read** can handle.

For files, directories, or special files with drivers that do not handle extended operations, the **readx** system call does exactly what the **read** system call does, and the *ext* parameter is ignored.

Each driver interprets the *ext* parameter in a device-dependent way, either as a value or as a pointer to a communication area. The nonextended **read** system call is equivalent to the extended **readx** system call with an *ext* parameter value of 0. Drivers must apply reasonable defaults when the *ext* parameter value is 0.

Return Value

Upon successful completion, the **read** and **readx** system calls return the number of bytes actually read and placed in the buffer; this number may be less than the value of the *nbyte* parameter if the file is associated with a communication line, or if the number of bytes left in the file is less than the value of the *nbyte* parameter. A value of 0 is returned when an end-of-file has been reached. (For information about communication files, see “ioctl” on page 2-54 and “termio” on page 6-103.) If **read** or **readx** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **read** and **readx** system calls fail if one or more of the following are true:

- | | |
|---------------|---|
| EBADF | <i>fildes</i> is not a valid file descriptor open for reading. |
| EAGAIN | An enforcement mode record lock is outstanding in the portion of the file that is to be read. |
| EFAULT | The <i>buf</i> parameter points to a location outside of the process’s allocated address space. |

read, readx

EDEADLK A deadlock would occur if the calling process were to sleep until the region to be read was unlocked.

EINTR A signal was caught during the **read** system call.

Related Information

In this book: “creat” on page 2-27, “dup” on page 2-30, “fcntl” on page 2-42, “ioctl” on page 2-54, “lockf” on page 2-62, “open” on page 2-88, “pipe” on page 2-93, and “termio” on page 6-103.

reboot

Purpose

Restarts the current virtual machine.

Syntax

```
int reboot (dev)  
char *dev;
```

```
int reboot ((char *) 0)  
  
int reboot ("VRM")
```

Description

The **reboot** system call restarts (re-IPLs) the current virtual machine from the block special file specified by the *dev* parameter. If the *dev* parameter is 0, the root device is assumed. The reboot is automatic and brings up **/unix** in the normal, nonmaintenance mode.

If the *dev* parameter is the character string "VRM", then Virtual Resource Manager is also restarted.

The effective user ID of the calling process must be superuser for this call to complete.

Warning: The **reboot** system call does not perform a **sync** operation or write pending output to disk. File systems may be damaged if **reboot** is invoked without first assuring that all disk output has completed.

Return Value

Upon successful completion, the **reboot** system call does not return. If the **reboot** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **reboot** system call fails if one or more of the following are true:

- | | |
|----------------|---|
| EPERM | The effective user ID of the calling process is not superuser. |
| ENOENT | The specified special file does not exist. |
| ENOTBLK | The <i>dev</i> parameter does not point to a block device file. |

reboot

ENXIO	The device associated with the <i>dev</i> parameter does not exist.
EFAULT	The <i>dev</i> parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “iplvm, waitvm” on page 2-56 and “sync” on page 2-157.

The **shutdown** command in *AIX Operating System Commands Reference*.

semctl

Purpose

Controls semaphore operations.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, val)
    — or —
int semctl (semid, semnum, cmd, buf)
    — or —
int semctl (semid, semnum, cmd, array)

int semid;
unsigned int semnum;
int cmd;
int val;
struct semid_ds *buf;
unsigned short array[ ];
```

Description

The **semctl** system call performs a variety of semaphore control operations as specified by the *cmd* parameter. The data type of the last parameter depends on the value of the *cmd* parameter. It is referred to as *val*, *buf*, or *array* to indicate one of the definitions given in the preceding **Syntax** section.

The first seven *cmds* get and set the values of a **sem** structure, which is defined in the **sys/sem.h** header file and contains the following members:

```
ushort  semval;    /* Operation permission structure */
short   sempid;    /* ID of last process that did a semop */
ushort  semncnt;   /* No. of processes awaiting semval > cval */
ushort  semzcnt;   /* No. of processes awaiting semval = 0 */
```

The following *cmds* are executed with respect to the semaphore specified by the *semid* and *semnum* parameters.

semctl

- GETVAL** Returns the value of **semval**, if the current process has read permission.
- SETVAL** Sets the value of **semval** to the value specified by *val*, if the current process has write permission. When this *cmd* is successfully executed, the **semadj** value corresponding to the specified semaphore is cleared in all processes.
- GETPID** Returns the value of **sempid**, if the current process has read permission.
- GETNCNT** Returns the value of **semmcnt**, if the current process has read permission.
- GETZCNT** Returns the value of **semzcnt**, if the current process has read permission.

The following *cmds* return and set every **semval** in the set of semaphores.

- GETALL** Stores **semvals** into the array pointed to by *array*, if the current process has read permission.
- SETALL** Sets **semvals** according to the array pointed to by *array*, if the current process has write permission. When this *cmd* is successfully executed, the **semadj** value corresponding to each specified semaphore is cleared in all processes.

The following *cmds* are also available:

- IPC_STAT** Stores the current value of each member of the data structure associated with the *semid* parameter into the structure pointed to by *buf*, if the current process has read permission. This structure is defined in **sys/sem.h** and contains the following members:

```
struct ipc_perm sem_perm; /* Operation permission structure */
struct sem *sem_base;    /* Pointer to first semaphore in set */
ushort      sem_nsems;   /* Number of semaphores in the set */
ushort      semlcnt;     /* Processes waiting on locked semaphore */
time_t      sem_otime;   /* Time of last semop call */
time_t      sem_ctime;   /* Time of the last change to this */
                        /* structure with a semctl call */
```

- IPC_SET** Sets the value of the following members of the data structure associated with the *semid* parameter to the corresponding value found in the structure pointed to by *buf*:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode /* Only the low-order nine bits */
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of **sem_perm.uid** in the data structure associated with the *semid* parameter.

IPC_RMID Removes the semaphore identifier specified by the *semid* parameter from the system and destroys the set of semaphores and data structures associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of **sem-perm.uid** in the data structure associated with the *semid* parameter.

Return Value

Upon successful completion, the value returned depends on the *cmd* parameter as follows:

<i>cmd</i>	Return Value
GETVAL	Returns the value of semval .
GETPID	Returns the value of sempid .
GETNCNT	Returns the value of semmcnt .
GETZCNT	Returns the value of semzcnt .
All others	Return a value of 0.

If **semctl** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **semctl** system call fails if one or more of the following are true:

EINVAL	The <i>semid</i> parameter is not a valid semaphore identifier.
EINVAL	The <i>semnum</i> parameter is less than 0 or greater than sem-nsems .
EINVAL	The <i>cmd</i> parameter is not a valid command.
EACCES	Operation permission is denied to the calling process.
ERANGE	The <i>cmd</i> parameter is SETVAL or SETALL and the value to which semval is to be set is greater than the system-imposed maximum.
EPERM	The <i>cmd</i> parameter is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal either to that of superuser or to the value of sem-perm.uid in the data structure associated with the <i>semid</i> parameter.
EFAULT	The <i>buf</i> or <i>array</i> parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “semget” on page 2-113 and “semop” on page 2-116.

semget

Purpose

Gets a set of semaphores.

Syntax

```
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

Description

The **semget** system call returns the semaphore identifier associated with the specified *key*. The *key* parameter is either the value **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm). See “ftok” on page 3-183 for details about this subroutine. The *nsems* parameter specifies the number of semaphores in the set.

The *semflg* parameter is constructed by logically OR-ing one or more of the following values:

IPC_CREAT	Creates the data structure if it does not already exist.
IPC_EXCL	Causes the semget system call to fail if IPC_CREAT is also set and the data structure already exists.
S_IRUSR	Permits the process that owns the data structure to read it.
S_IWUSR	Permits the process that owns the data structure to modify it.
S_IRGRP	Permits the group associated with the data structure to read it.
S_IWGRP	Permits the group associated with the data structure to modify it.
S_IROTH	Permits others to read the data structure.
S_IWOTH	Permits others to modify the data structure.

The values that begin with **S-I-** are defined in the **sys/stat.h** header file and are a subset of the access permissions that apply to files.

semget

The **semget** system call creates a data structure for the semaphore ID and an array containing *nsems* semaphores if one of the following is true:

- The *key* parameter is equal to **IPC_PRIVATE**.
- The *key* parameter does not already have a semaphore identifier associated with it, and **IPC_CREAT** is set.

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

- **sem_perm.cuid** and **sem_perm.uid** are set equal to the effective user ID of the calling process.
- **sem_perm.cgid** and **sem_perm.gid** are set equal to the effective group ID of the calling process.
- The low-order nine bits of **sem_perm.mode** are set equal to the low-order nine bits of the *semflg* parameter.
- **sem_nsems** is set equal to the value of the *nsems* parameter.
- **sem_otime** is set equal to 0 and **sem_ctime** is set equal to the current time.

If the *key* parameter is not **IPC_PRIVATE**, **IPC_EXCL** is not set, and a semaphore identifier already exists for the specified *key*, then the value of the *nsems* parameter specifies the number of semaphores that the current process needs. If the *nsems* parameter is 0, then any number of semaphores is acceptable. If the *nsems* parameter is not 0, then the **semget** system call fails if the set contains fewer than *nsems* semaphores.

Return Value

Upon successful completion, a semaphore identifier is returned. If **semget** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **semget** system call fails if one or more of the following are true:

- | | |
|---------------|---|
| EINVAL | The <i>nsems</i> parameter is less than 0, equal to 0, or greater than the system-imposed limit. |
| EACCES | A semaphore identifier exists for the <i>key</i> parameter but operation permission, as specified by the low-order nine bits of the <i>semflg</i> parameter, is not granted. |
| EINVAL | A semaphore identifier exists for the <i>key</i> parameter, but the number of semaphores in the set associated with it is less than the value of the <i>nsems</i> parameter and the <i>nsems</i> parameter is not equal to 0. |

- ENOENT** A semaphore identifier does not exist for the *key* parameter and **IPC_CREAT** is not set.
- ENOSPC** A semaphore identifier is to be created, but doing so would exceed the maximum number of identifiers allowed system wide.
- EEXIST** A semaphore identifier exists for the *key* parameter, and both **IPC_CREAT** and **IPC_EXCL** are set.

Related Information

In this book: “semctl” on page 2-109, “semop” on page 2-116, and “ftok” on page 3-183.

semop

semop

Purpose

Performs semaphore operations.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf sops[ ];
unsigned int nsops;
```

Description

The **semop** system call performs operations on the set of semaphores associated with the semaphore identifier specified by the *semid* parameter. The *sops* parameter points to an array of structures, each of which specifies a semaphore operation. The *nsops* parameter is the number of such structures in the array. The **sembuf** structure is defined in the **sys/sem.h** header file, and it contains the following members:

```
ushort  sem_num; /* Semaphore number */
short   sem_op;  /* Semaphore operation */
short   sem_flg; /* Operation flags */
```

Each semaphore operation specified by a **sem_op** is performed on the corresponding semaphore specified by *semid* and **sem_num**. The **sem_flg** for each operation is either 0, or is constructed by logically OR-ing one or more of the following values:

SEM_UNDO	Specifies whether to modify semadj values.
SEM_ORDER	Specifies whether to perform the operations atomically or individually. (This applies only to the sem_flg of the first operation specified in the <i>sops</i> array.)
IPC_NOWAIT	Specifies whether to wait or to return immediately when a semaphore's semval is not a certain value.

If **SEM_ORDER** is not set in *sops*[0].**sem_flg** (the default), then all of the semaphore operations specified in the *sops* array are performed atomically. This means that no

semval value for any **sem-num** that appears in the entire array of operations is modified until all the semaphore operations can be completed. If the calling process must wait until some **semval** requirement is met, then the **semop** system call does so before performing any of the operations. If any semaphore operation would cause an error to occur, then none of the operations are performed.

If **SEM-ORDER** is set in **sops[0].sem_flg**, then the operations are performed individually in the order that they appear in the **sops** array, regardless of whether any of the operations require the process to wait. If an operation encounters an error condition, then the **semop** system call sets **SEM-ERR** in the **sem_flg** of the failing operation, sets **errno** to indicate the error, and returns a value of -1. In this case, the operations that precede the failing one in the **sops** array have been performed, but those following it have not.

The action taken for **SEM-UNDO** and **IPC-NOWAIT** is described in the following text.

The **sem-op** field of the **sembuf** structure specifies one of the following three semaphore operations:

1. If **sem-op** is a positive integer and the current process has write permission, then the value of **sem-op** is added to **semval**. If **SEM-UNDO** is set in **sem_flg**, then the value of **sem-op** is also subtracted from the calling process's **semadj** value for the specified semaphore.
2. If **sem-op** is a negative integer and the current process has write permission, then one of the following occurs:
 - If **semval** is greater than or equal to the absolute value of **sem-op**, the absolute value of **sem-op** is subtracted from **semval**. Also, if **SEM-UNDO** is set in **sem_flg**, the absolute value of **sem-op** is added to the calling process's **semadj** value for the specified semaphore. The **exit** system call adds the **semadj** value to the semaphore's **semval** when the process terminates (see "exit, _exit" on page 2-38).
 - If **semval** is less than the absolute value of **sem-op** and **IPC-NOWAIT** is set in **sem_flg**, **semop** returns a value of -1 and sets **errno** to **EAGAIN**.
 - If **semval** is less than the absolute value of **sem-op** and **IPC-NOWAIT** is not set in **sem_flg**, then **semop** increments the **semncnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:
 - **semval** becomes greater than or equal to the absolute value of **sem-op**. When this occurs, the value of **semncnt** associated with the specified semaphore is decremented, the absolute value of **sem-op** is subtracted from **semval** and, if **SEM-UNDO** is set in **sem_flg**, the absolute value of **sem-op** is added to the calling process's **semadj** value for the specified semaphore.
 - The **semid** for which the calling process is awaiting action is removed from the system (see "semctl" on page 2-109). When this occurs, **errno** is set equal to **EIDRM**, and a value of -1 is returned.

semop

- The calling process receives a signal that is to be caught. When this occurs, the value of **semncnt** associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in the **signal** system call.
- 3. If **sem-op** is 0 and the current process has read permission, then one of the following occurs:
 - If **semval** is 0, then **semop** returns a value of 0.
 - If **semval** is not equal to 0 and **IPC_NOWAIT** is set in **sem_flg**, then **semop** returns a value of -1 and sets **errno** to **EAGAIN**.
 - If **semval** is not equal to 0 and **IPC_NOWAIT** is not set in **sem_flg**, **semop** increments the **semzcnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:
 - **semval** becomes 0, at which time the value of **semzcnt** associated with the specified semaphore is decremented.
 - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, **errno** is set equal to **EIDRM**, and a value of -1 is returned.
 - The calling process receives a signal that is to be caught. When this occurs, the value of **semzcnt** associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in the **signal** system call.

Return Value

Upon successful completion, the **semop** system call returns a value of 0. Also, the **sempid** value for each semaphore that is operated upon is set to the process ID of the calling process.

If **semop** fails, a value of -1 is returned and **errno** is set to indicate the error. If **SEM_ORDER** was set in the **sem_flg** for the first semaphore operation in the *sops* array, then **SEM_ERR** is set in the **sem_flg** for the failing operation.

Diagnostics

The **semop** system call fails if one or more of the following are true for any of the semaphore operations specified by the *sops* parameter. If the operations were performed individually, then see the preceding discussion of **SEM_ORDER** for more information about error situations.

EINVAL The *semid* parameter is not a valid semaphore identifier.

EFBIG	sem_num is less than 0 or it is greater than or equal to the number of semaphores in the set associated with the <i>semid</i> parameter.
E2BIG	The <i>nsops</i> parameter is greater than the system-imposed maximum.
EACCES	Operation permission is denied to the calling process.
EAGAIN	The operation would result in suspension of the calling process, but IPC_NOWAIT is set in sem_flg
ENOSPC	The limit on the number of individual processes requesting a SEM_UNDO would be exceeded.
EINVAL	The number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the limit.
ERANGE	An operation would cause a semval to overflow the system-imposed limit.
ERANGE	An operation would cause a semadj value to overflow the system-imposed limit.
EFAULT	The <i>sops</i> parameter points to a location outside of the process's allocated address space.
EINTR	The semop system call received a signal.
EIDRM	The semaphore identifier <i>semid</i> has been removed from the system.

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “exit, _exit” on page 2-38, “fork” on page 2-44, “semctl” on page 2-109, and “semget” on page 2-113.

setgroups

setgroups

Purpose

Sets the group access list.

Syntax

```
#include <grp.h>
```

```
int setgroups (ngroups, gidset);  
int ngroups, *gidset;
```

Description

The **setgroups** system call sets the group access list of the current user process according to the array pointed to by the *gidset* parameter. The *ngroups* parameter indicates the number of entries in the array and must not be more than **NGROUPS**, as defined in the **grp.h** header file. Only a process with an effective user ID of superuser can set new groups.

Return Value

Upon successful completion, a value of 0 is returned. If the **setgroups** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **setgroups** system call fails if one or more of the following is true:

- | | |
|---------------|--|
| EPERM | The caller is not superuser. |
| EINVAL | The value of the <i>ngroups</i> parameter is greater than NGROUPS . |
| EFAULT | The <i>gidset</i> parameter points to a location outside of the process's allocated address space. |

Related Information

In this book: “getgroups” on page 2-50 and “initgroups” on page 3-214.

setpgrp

setpgrp

Purpose

Sets process group ID.

Syntax

```
int setpgrp ( )
```

Description

The **setpgrp** system call sets the process group ID of the calling process to the process ID of the calling process.

Return Value

Upon successful completion, the new process group ID is returned.

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “fork” on page 2-44, “getpid, getpgrp, getppid” on page 2-52, “kill” on page 2-58, and “signal” on page 2-139.

setuid, setgid

Purpose

Sets a process's user and groups IDs.

Syntax

```
int setuid (uid)  
int uid;
```

```
int setgid (gid)  
int gid;
```

Description

The **setuid** system call sets the real and effective user IDs of the calling process. If the effective user ID of the calling process is superuser, then the real and effective user IDs are set to the value of the *uid* parameter. If the effective user ID of the calling process is not superuser, but the real user ID is equal to the value of the *uid* parameter, or the process's original effective user ID as set by the **exec** system call is equal to *uid*, then the effective user ID is set to the value of the *uid* parameter.

The **setgid** system call sets the real and effective group IDs of the calling process. If the effective user ID of the calling process is superuser, then the real and effective group IDs are set to the value of the *gid* parameter. If the effective user ID of the calling process is not superuser, but the real group ID is equal to the value of the *gid* parameter, or the process's original effective group ID as set by the **exec** system call is equal to *gid*, then the effective group ID is set to the value of the *gid* parameter.

Return Value

Upon successful completion, a value of 0 is returned. If the **setuid** or **setgid** system call fails, then a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **setuid** and **setgid** system calls fail if the following is true:

EPERM	The <i>uid</i> (<i>gid</i>) parameter is not equal to the real user (group) ID of the process or to the original effective user (group) ID as set by the exec system call, and the effective user ID is not superuser.
--------------	---

setuid, setgid

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “getpid, getpgrp, getppid” on page 2-52, and “getuid, geteuid, getgid, getegid” on page 2-53.

shmat

Purpose

Attaches a shared memory segment or a mapped file to the current process.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;
```

Description

The **shmat** system call attaches the shared memory segment or mapped file associated with the shared memory identifier (returned by **shmget**) or file descriptor (returned by **open**) specified by the *shmid* parameter to the address space of the calling process.

The segment or file is attached at the address specified by the *shmaddr* parameter as follows:

- If the *shmaddr* parameter is equal to 0, the segment or file is attached at the first available address as selected by the system.
- If the *shmaddr* parameter is **not** equal to 0, and **SHM_RND** is set in *shmflg*, the segment or file is attached at the next lower segment boundary. This address given by (*shmaddr* - (*shmaddr* modulo **SHMLBA**)).
- If the *shmaddr* parameter is *not* equal to 0 and **SHM_RND** *not* set in *shmflg*, the segment or file is attached at the address given by the *shmaddr* parameter. If this address does not point to a segment boundary, then the **shmat** system call returns the value -1 and sets **errno** to **EINVAL**.

The *shmflg* parameter specifies several options. Its value is either 0, or is constructed by logically OR-ing one or more of the following values:

SHM_RND	Rounds the address given by the <i>shmaddr</i> parameter to the next lower segment boundary, if necessary.
----------------	--

- SHM_MAP** Maps a file onto the address space instead of a shared memory segment. The *shmid* must specify an open file descriptor in this case.
- SHM_RDONLY** Specifies read-only mode instead of the default read-write mode.
- SHM_COPY** Maps a file in copy-on-write mode.

Either **SHM_RDONLY** or **SHM_COPY** may be specified, but not both.

If **SHM_MAP** is not set in *shmflg*, then a shared memory segment is attached to the data segment. It is attached for reading if **SHM_RDONLY** is set in *shmflg* and if the current process has read permission. If **SHM_RDONLY** is not set and the current process has both read and write permission, then it is attached for reading and writing.

If **SHM_MAP** is set in *shmflg*, then file mapping takes place. In this case, the **shmat** system call maps the file open on file descriptor *shmid* onto a segment. The file must be a regular file. The segment is then mapped into the process's address space.

When file mapping is requested, the *shmflg* parameter specifies how the file is to be mapped. If **SHM_RDONLY** is set, then the file is mapped read-only. If **SHM_COPY** is set, then the file is mapped copy-on-write. If neither of these cases is true, then the file is mapped read-write. The file must be opened for writing before it can be mapped read-write or copy-on-write.

All processes that map the same file read-only or read-write map to the same segment. This segment remains mapped until the last process mapping the file closes it.

All processes that map the same file copy-on-write map the same copy-on-write segment. Changes to the shared segment do not affect the contents of the file resident in the file system until an **fsync** system call is issued for a file descriptor for which copy-on-write mapping was requested. If a process requests copy-on-write mapping for a file and the copy-on-write segment does not yet exist, then it is created, and that segment is maintained for sharing until the last process attached to it detaches it with a **close** system call. When the mapped file is closed, the segment is detached. The next request for copy-on-write mapping for the same file causes a new segment to be created for the file.

A file descriptor can be used to map the corresponding file only once. A file can be multiply mapped by using multiple file descriptors. However, a file cannot be mapped both read-write and copy-on-write by one or more users at the same time. The results are unpredictable if a file that one process has mapped copy-on-write is modified by another process with the **write** system call, unless that process has also attached the copy-on-write segment with a **shmat** system call.

When a file is mapped onto a segment, the file is referenced by accessing the segment. The memory paging system automatically takes care of the physical I/O. References beyond the end of the file cause the file to be extended in increments of the page size.

Return Value

Upon successful completion, the segment start address of the attached shared memory segment or mapped file is returned. If **shmat** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **shmat** system call fails and the shared memory segment or mapped file is not attached if one or more of the following are true:

- | | |
|----------------|--|
| EINVAL | The <i>shmid</i> parameter is not a valid shared memory identifier. |
| EACCES | Operation permission is denied to the calling process. |
| ENOMEM | The available data space in memory is not large enough to hold the shared memory segment. |
| ENOMEM | The available data space in memory is not large enough to hold the mapped file data structure. |
| EINVAL | The <i>shmaddr</i> parameter is not equal to 0, and the value of (<i>shmaddr</i> - (<i>shmaddr</i> modulo SHMLBA)) points to a location outside of the process's allocated address space. |
| EINVAL | The <i>shmaddr</i> parameter is not equal to 0, SHM-RND is not set in <i>shmflg</i> , and the the <i>shmaddr</i> parameter points to a location outside of the process's allocated address space. |
| EINVAL | The <i>shmaddr</i> parameter is not equal to 0, SHM-RND is not set in <i>shmflg</i> , and the the <i>shmaddr</i> parameter does not point to a segment boundary. |
| EEXIST | The file to be mapped has already been mapped. |
| ETXTBSY | The shmat system call attempted to map a file onto a segment attached to a shared library. |
| EMFILE | The number of shared memory segments attached to the calling process would exceed the system-imposed limit. |
| EBADF | A file descriptor to map does not refer to an open regular file, or both read-only and copy-on-write modes were requested. |
| EACCES | A file to be mapped is open read-only, but the segment is to be mapped read-write or copy-on-write. |
| EACCES | The file is to be mapped read-write, but the file is currently mapped copy-on-write; or the file is to be mapped copy-on-write, but it is currently mapped read-write. |

shmat

- EACCES** The file to be mapped has enforced locking enabled, and the file is currently locked.
- EFBIG** The file to be mapped is larger than the maximum size of a segment.

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “exit, _exit” on page 2-38, “fclear” on page 2-40, “fork” on page 2-44, “fsync” on page 2-46, “ftruncate” on page 2-48, “read, readx” on page 2-104, “shmctl” on page 2-129, “shmget” on page 2-134, “shmdt” on page 2-132, and “write, writex” on page 2-178.

shmctl

Purpose

Controls shared memory operations.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(shmid, cmd, buf)
int shmid, cmd;
struct shmids *buf;
```

Description

The **shmctl** system call performs a variety of shared memory control operations as specified by the *cmd* parameter. The *shmid* parameter is a shared memory identifier returned by the **shmget** system call. The following *cmds* are available:

IPC_STAT Places the current value of each member of the data structure associated with the *shmid* parameter into the **shmids** structure pointed to by the *buf* parameter. The current process must have read permission in order to perform this operation. The **shmids** structure is defined in the **sys/shm.h** header file, and it contains the following members:

```
struct ipc_perm shm_perm; /* Operation permission structure */
int shm_segsz; /* Segment size */
ushort shm_segid; /* Segment identifier */
ushort shm_lpid; /* ID of last process to call shmop */
ushort shm_cpid; /* ID of process that created this shmid */
ushort shm_nattch; /* Current number of processes attached */
ushort shm_cnattch; /* No. of in-memory processes attached */
time_t shm_atime; /* Time of last shmat call */
time_t shm_dtime; /* Time of last shmdt call */
time_t shm_ctime; /* Time of the last change to this */
/* structure with a shmctl call */
```

shmctl

IPC_SET Sets the value of the following members of the data structure associated with the *shmid* parameter to the corresponding value found in the structure pointed to by the *buf* parameter:

```
shm_perm.uid  
shm_perm.gid  
shm_perm.mode /* Only the low-order nine bits */
```

This *cmd* can only be performed by a process that has an effective user ID equal to either that of superuser or to the value of **shm_perm.uid** in the data structure associated with the *shmid* parameter.

IPC_RMID Removes the shared memory identifier specified by the *shmid* parameter from the system and erases the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of **shm_perm.uid** in the data structure associated with the *shmid* parameter.

Return Value

Upon successful completion, a value of 0 is returned. If **shmctl** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **shmctl** system call fails if one or more of the following are true:

- | | |
|---------------|--|
| EINVAL | The <i>shmid</i> parameter is not a valid shared memory identifier. |
| EINVAL | The <i>cmd</i> parameter is not a valid command. |
| EACCES | The <i>cmd</i> parameter is equal to IPC_STAT and read permission is denied to the calling process. |
| EPERM | The <i>cmd</i> parameter is equal to IPC_RMID or IPC_SET , and the effective user ID of the calling process is neither equal to the superuser ID, nor is it equal to the value of shm_perm.uid in the data structure associated with <i>shmid</i> . |
| EFAULT | The <i>buf</i> parameter points to a location outside of the process's allocated address space. |

Related Information

In this book: “shmat” on page 2-125, “shmdt” on page 2-132, and “shmget” on page 2-134.

shmdt

shmdt

Purpose

Detaches a shared memory segment.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmdt (shmaddr)
char *shmaddr;
```

Description

The **shmdt** system call detaches, from the calling process's data segment, the shared memory segment located at the address specified by the *shmaddr* parameter.

Mapped file segments are automatically detached when no longer in use. However, you can use the **shmdt** system call to explicitly release the segment register used to map a file. Shared memory segments must be explicitly detached with **shmdt**.

Return Value

Upon successful completion, a value of 0 is returned. If **shmdt** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **shmdt** system call fails and the shared memory segment is not detached if the following is true:

- | | |
|----------------|--|
| EINVAL | The <i>shmaddr</i> parameter is not the data segment start address of a shared memory segment. |
| ETXTBSY | The shmdt system call attempted to detach a segment attached to a shared library. |

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “exit, _exit” on page 2-38, “fclear” on page 2-40, “fork” on page 2-44, “fsync” on page 2-46, “shmctl” on page 2-129, “shmget” on page 2-134, and “shmat” on page 2-125.

shmget

shmget

Purpose

Gets shared memory segment.

Syntax

```
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

Description

The **shmget** system call returns the shared memory identifier associated with the specified *key*. The *key* parameter is either the value **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm). See “ftok” on page 3-183 for details about this subroutine. The *size* parameter specifies the number of bytes of shared memory required.

The *shmflg* parameter is constructed by logically OR-ing one or more of the following values:

IPC_CREAT	Creates the data structure if it does not already exist.
IPC_EXCL	Causes the shmget system call to fail if IPC_CREAT is also set and the data structure already exists.
S_IRUSR	Permits the process that owns the data structure to read it.
S_IWUSR	Permits the process that owns the data structure to modify it.
S_IRGRP	Permits the group associated with the data structure to read it.
S_IWGRP	Permits the group associated with the data structure to modify it.
S_IROTH	Permits others to read the data structure.
S_IWOTH	Permits others to modify the data structure.

The values that begin with **S-I-** are defined in the **sys/stat.h** header file and are a subset of the access permissions that apply to files.

A shared memory identifier, its associated data structure, and a shared memory segment equal in bytes to the value of the *size* parameter are created for the *key* parameter if one of the following is true:

- The *key* parameter is equal to **IPC_PRIVATE**.
- The *key* parameter does not already have a shared memory identifier associated with it, and **IPC_CREAT** is set.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- **shm_perm.cuid** and **shm_perm.uid** are set equal to the effective user ID of the calling process.
- **shm_perm.cgid** and **shm_perm.gid** are set equal to the effective group ID of the calling process.
- The low-order nine bits of **shm_perm.mode** are set equal to the low-order nine bits of the *shmflg* parameter.
- **shm-segsz** is set equal to the value of the *size* parameter.
- **shm-lpid**, **shm-nattch**, **shm-atime**, and **shm-dtime** are set equal to 0.
- **shm-ctime** is set equal to the current time.

Return Value

Upon successful completion, a shared memory identifier is returned. If **shmget** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **shmget** system call fails if one or more of the following are true:

- | | |
|---------------|---|
| EINVAL | The <i>size</i> parameter is less than the system-imposed minimum or greater than the system-imposed maximum. |
| EACCES | A shared memory identifier exists for the <i>key</i> parameter but operation permission as specified by the low-order nine bits of the <i>shmflg</i> parameter is not granted. |
| EINVAL | A shared memory identifier exists for <i>key</i> , but the size of the segment associated with it is less than the <i>size</i> parameter and the <i>size</i> parameter is not equal to 0. |
| ENOENT | A shared memory identifier does not exist for the <i>key</i> parameter and IPC_CREAT not set. |

shmget

- | | |
|---------------|--|
| ENOSPC | A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide will be exceeded. |
| ENOMEM | A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request. |
| EEXIST | A shared memory identifier exists for the <i>key</i> parameter, and both IPC_CREAT and IPC_EXCL are set. |

Related Information

In this book: “shmat” on page 2-125, “shmctl” on page 2-129, “shmdt” on page 2-132, and “ftok” on page 3-183.

sigblock

Purpose

Blocks signals.

Syntax

```
int sigblock (mask)  
int mask;
```

Description

The **sigblock** system call causes the signals specified by the *mask* parameter to be added to the set of signals currently being blocked from delivery. The signals are blocked from delivery by logically OR-ing the *mask* parameter into the process's signal mask. Signal *i* is blocked if the *i*-th bit in *mask* is a 1.

It is not possible to block **SIGKILL**. The system provides no indication of this restriction.

Typically, the **sigblock** system call is used to block signals during a critical section of code, and then **sigsetmask** is called to restore the mask to the previous value returned by **sigblock**.

Return Value

Upon completion, the previous set of masked signals is returned.

Example

The following example sets the signal mask to block **SIGINT** from delivery, in addition to the signals that are already blocked:

```
#include <signal.h>  
  
long prevmask;  
...  
prevmask = sigblock ((long) 1 << (SIGINT - 1));
```


sigblock

Related Information

In this book: “kill” on page 2-58, “signal” on page 2-139, “sigvec” on page 2-150, and “sigsetmask” on page 2-146.

signal

Purpose

Specifies the action to take upon receipt of a signal.

Syntax

```
#include <sys/signal.h>

int (*signal (sig, action)) ( )
int sig;
void (*action) ( );
```

Description

The **signal** system call allows the calling process to choose one of three ways to handle the receipt of a specific signal. The *sig* parameter specifies the signal and the *action* parameter specifies the choice.

The *sig* parameter can be any one of the following signal values except **SIGKILL**. Each of the names shown below is defined in the **sys/signal.h** header file with the value of the corresponding signal number.

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGILL	4*	Illegal instruction (not reset when caught)
SIGTRAP	5*	Trace trap (not reset when caught)
SIGIOT	6*	Abort process (see “abort” on page 3-5)
SIGDANGER	7 + %	The system is likely to “crash” soon
SIGFPE	8*	Arithmetic exception, integer divide by 0, or floating point exception
SIGKILL	9	Kill (cannot be caught or ignored)
SIGBUS	10*	Specification exception
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad parameter to system call
SIGPIPE	13	Write on a pipe when there is no process to read it

signal

SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGUSR1	16	User-defined signal 1
SIGUSR2	17	User-defined signal 2
SIGCLD	18 +	Death of a child process
SIGPWR	19 +	Power-fail restart (not reset when caught)
SIGAIO	25	Basic LAN signal for asynchronous I/O
SIGPTY	26	PTY device driver read/write availability
SIGIOINT	27	I/O intervention required
SIGGRANT	28#	HFT monitor access wanted
SIGRETRACT	29#	HFT monitor access should be relinquished
SIGSOUND	30#	An HFT sound control has completed execution
SIGMSG	31#	Input data has been stored into the HFT monitor mode ring buffer

The symbols in the preceding table have the following meaning:

- * A memory image file (**core** file) is created when one of these signals is received. This is explained in more detail in the following discussion of **SIG_DFL**.
- + These signals require special consideration, as described in “Special Signals” on page 2-142.
- % See the **sys/robust.h** header file for the conditions that can cause this signal. The most likely cause is a shortage of paging space (**PGSDANGER**). Also see the **pslotwarn**, **pslotkill**, and **pslotpanic** keywords in “master” on page 4-93.
- # For more information on the use of these signals, see “hft” on page 6-19.

The *action* parameter is one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values of are as follows:

SIG_DFL — Default action: Terminate process upon receipt of signal.

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in the **exit** system call. In addition, a *memory image* file will be created in the current directory of the receiving process if *sig* is one for which an asterisk appears in the preceding list *and* the following conditions are met:

- The effective user ID and the real user ID of the receiving process are equal.
- An ordinary file named **core** exists in the current directory and is writable, or it can be created. If the file must be created, it will have the following properties:
 - The access permission code 0666 (0x1B6), modified by the file creation mask (see “umask” on page 2-163)

- A file owner ID that is the same as the effective user ID of the receiving process
- A file group ID that is the same as the effective group ID of the receiving process.

SIG_IGN — Ignore signal.

The signal *sig* is to be ignored.

Note: The **SIGKILL** signal cannot be ignored.

function address — Catch signal.

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by the *action* parameter. The signal number *sig* is passed as the only parameter to the signal-catching function. Before calling the signal-catching function, the value of *action* for the caught signal is set to **SIG_DFL** unless the signal is **SIGILL**, **SIGTRAP**, or **SIGPWR**.

Upon return from the signal-catching function, the previous value of the *action* for the signal is restored, and the receiving process resumes execution at the point at which it was interrupted.

Note that after a signal is received, there is a period of time during which the signal action is set to **SIG_DFL** and the signal-catching function has not had a chance to re-establish itself as the catcher for this signal. If the signal occurs again during that period, it will not be caught. The **sigvec** system call offers an enhanced signal-handling capacity to avoid this *race condition*.

When a signal that is to be caught occurs during a **read**, **write**, **open**, or **ioctl** system call on a slow device (like a terminal; but not an ordinary file), during a **pause** system call, or during a **wait** system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal-catching function will be executed and then the interrupted system call will return a -1 to the calling process with **errno** set to **EINTR**.

Note: The **SIGKILL** signal cannot be caught.

Warning: The **signal** system call does not check the validity of the *action* parameter. If it points to a location outside of the process's allocated address space, then the process receives a memory fault when the system attempts to call the signal handler. If *action* points to anything other than a subroutine, the results are unpredictable.

Special Signals

Some signals are handled differently from those described previously. These signals are:

SIGDANGER	7+	The system is likely to “crash” soon.
SIGCLD	18+	Death of a child process
SIGPWR	19+	Power-fail restart (not reset when caught)

For **SIGDANGER** and **SIGPWR**, the actions prescribed by the *action* parameter are as follows:

SIG_DFL The signal is ignored.

SIG_IGN The signal is ignored.

function address The signal-catching function pointed to by *action* is called.

For **SIGCLD**, the actions prescribed by the *action* parameter are as follows:

SIG_DFL The signal is ignored.

SIG_IGN The signal is ignored. Also, the child processes of the calling process do not create zombie processes when they terminate. (See “exit, _exit” on page 2-38 for more information about zombie processes.)

function address The signal-catching function pointed to by *action* is called. Any **SIGCLD** signals received while the signal-catching function is running are queued.

The setting of the *action* for the **SIGCLD** signal affects the **wait** and **exit** system calls in the following ways:

wait If the *action* value of **SIGCLD** is set to **SIG_IGN** and a **wait** system call is executed, the **wait** blocks until all of the child processes of the calling process terminate. It then returns a value of -1 with **errno** set to **ECHILD**.

exit If, in the parent of the exiting process, the *action* value of **SIGCLD** is set to **SIG_IGN**, then the exiting process does not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that can be piped into in this manner, and thus become the parent of other processes, should not set **SIGCLD** to be caught. Otherwise, it will receive unexpected **SIGCLD** signals.

After a **fork** system call, the child process inherits all signals from its parent.

The **exec** system calls reset all caught signals to the default action. Signals that cause the default action continue to do so. Ignored signals continue to be ignored.

Return Value

Upon successful completion, *signal* returns the previous value of *action* for the specified signal *sig*. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **signal** system call fails if the following is true:

EINVAL The *sig* parameter is not a valid signal number, or it is **SIGKILL**.

Related Information

In this book: “acct” on page 2-11, “exit, _exit” on page 2-38, “kill” on page 2-58, “pause” on page 2-92, “ptrace” on page 2-100, “sigblock” on page 2-137, “sigpause” on page 2-144, “sigsetmask” on page 2-146, “sigstack” on page 2-148, “sigvec” on page 2-150, “umask” on page 2-163, “wait” on page 2-176, “setjmp, longjmp” on page 3-287, and “core” on page 4-38.

sigpause

sigpause

Purpose

Atomically releases blocked signals and waits for an interrupt.

Syntax

```
int sigpause (sigmask)
int sigmask;
```

Description

The **sigpause** system call sets the process's signal mask to the value of the *sigmask* parameter and then and waits for a signal to arrive. Upon return, the previous signal mask is restored. The **sigpause** system call terminates by being interrupted, returning -1, and setting **errno** to **EINTR**.

The **sigpause** system call sets the signal mask and waits for an interrupt as one **atomic operation**. This means that signals cannot occur between the operations of setting the mask and waiting for a signal. If a program invokes the **sigsetmask** and **pause** system calls separately, then a signal that occurs between these system calls might not be noticed by **pause**.

In normal usage, a signal is blocked by using the **sigblock** system call at the beginning of a critical section. The process then determines whether there is work for it to do. If no work is to be done, then the process waits for work by calling **sigpause** with the mask previously returned by **sigblock**.

Return Value

If the signal is caught by the calling process and control is returned from the signal handler, then the calling process resumes execution after the **sigpause** system call, which always returns a value of -1 and sets **errno** to **EINTR**.

Related Information

In this book: “pause” on page 2-92, “sigblock” on page 2-137, “signal” on page 2-139, “sigsetmask” on page 2-146, and “sigvec” on page 2-150.

sigsetmask

sigsetmask

Purpose

Sets the current signal mask.

Syntax

```
int sigsetmask (mask)  
int mask;
```

Description

The **sigsetmask** system call sets the current signal mask as specified by the *mask* parameter. The signal mask determines which signals is blocked from delivery to the process. Signal *i* is blocked if the *i*-th bit in *mask* is a 1.

Typically, you would use the **sigblock** system call to block signals during a critical section of code and then use the **sigsetmask** system call to restore the mask to the previous value returned by the **sigblock** system call.

The **sigsetmask** system call does not allow **SIGKILL** to be blocked. If a program attempts to block **SIGKILL**, **sigsetmask** gives no indication of the error.

Return Value

Upon successful completion, the previous set of masked signals is returned.

Example

To set the signal mask to block only **SIGINT** from delivery:

```
#include <signal.h>
```

```
long prevmask;
```

```
...  
prevmask = sigsetmask ((long) 1 << (SIGINT - 1));
```


Related Information

In this book: “kill” on page 2-58, “signal” on page 2-139, “sigvec” on page 2-150, “sigblock” on page 2-137, and “sigpause” on page 2-144.

sigstack

sigstack

Purpose

Sets and gets signal stack context.

Syntax

```
#include <signal.h>
```

```
int sigstack (instack, outstack)  
struct sigstack *instack, *outstack;
```

Description

The **sigstack** system call defines an alternate stack on which signals are to be processed.

If the value of the *instack* parameter is nonzero, then it points to a **sigstack** structure, which has the following members:

```
    caddr_t    ss_sp;  
    int        ss_onstack;
```

The value of *instack->ss_sp* specifies the stack pointer of the new signal stack. Since stacks grow from numerically greater addresses to lower ones, the stack pointer passed to the **sigstack** system call should point to the numerically high end of the stack area to be used. *instack->ss_onstack* should be set to 1 if the process is currently executing on that stack; otherwise, it should be 0.

If the value of the *outstack* parameter is nonzero, then it points to a **sigstack** structure into which the **sigstack** system call stores the current signal stack state.

If the value of the *instack* parameter is 0 (that is, a **NULL** pointer), then the signal stack state is not set. If the value of the *outstack* parameter is 0, then the previous signal stack state is not reported.

When a signal occurs whose handler is to run on the signal stack, the system checks to see if the process is already executing on that stack. If so, then it continues to do so even after the handler returns. If not, then the signal handler runs on the signal stack, and the original stack is restored when the handler returns.

Use the **sigvec** system call to specify whether or not a given signal's handler routine is to run on the signal stack.

Warning: A signal stack does not automatically increase in size as a normal stack does. If the stack overflows, unpredictable results may occur.

Return Value

Upon successful completion, a value of 0 is returned. If the **sigstack** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **sigstack** system call fails and the signal stack context remains unchanged if the following is true:

EFAULT The *instack* or *outstack* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “signal” on page 2-139, “sigvec” on page 2-150, and “setjmp, longjmp” on page 3-287.

sigvec

sigvec

Purpose

Selects enhanced signal facilities.

Syntax

```
#include <sys/signal.h>

int sigvec (sig, invec, outvec)
int sig;
struct sigvec *invec, *outvec;
```

Description

The **sigvec** system call allows the user to select standard or enhanced signal-handling facilities. Like the **signal** system call, it sets the action to take upon the receipt of a signal, but it also sets additional features.

The **sigvec** system call assigns a handler for a specific signal. If the *invec* parameter is nonzero, it points to a **sigvec** structure that specifies a handler routine and mask to be used when delivering the specified signal. The **sigvec** structure has the following members:

```
int (*sv_handler) ( );
int sv_mask;
int sv_onstack;
```

If the **SIG-STK** bit of **sv-onstack** is set, then the system runs the handler on the signal stack specified by the **sigstack** system call. If this bit is not set, then the handler executes on the stack of the interrupted process. If the **SIG-STD** bit of **sv-onstack** is set, then standard signal processing is used. If this bit is not set, then enhanced signal processing is used.

The default action for a signal can be reinstated by setting **sv-handler** to **SIG-DFL**. If **sv-handler** is set to **SIG-IGN**, then the signal is ignored, and pending instances of the signal are discarded. See “signal” on page 2-139 for a detailed description of the default signal actions.

If the *outvec* parameter is nonzero, then the previous handling information for the signal is stored in the **sigvec** structure pointed to by *outvec*.

If the value of the *invec* parameter is 0 (that is, a **NULL** pointer), then the signal handler information is not set. If the value of the *outvec* parameter is 0, then the previous signal handler information is not reported.

Once a signal handler is assigned, it remains assigned until another **sigvec**, **signal**, or **exec** system call is made.

Warning: The **sigvec** system call does not check the validity of the **sv_handler** pointer. If it points to a location outside of the process's allocated address space, then the process receives a memory fault when the system attempts to call the signal handler.

The signal-handler subroutine can be declared as follows:

```
handler (sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

The *sig* parameter is the signal number. The *code* parameter is provided only for compatibility with other UNIX-compatible systems, and its value is always 0. The *scp* parameter points to the **sigcontext** structure that is later used to restore the process's previous execution context. The **sigcontext** structure is defined in **signal.h**.

Return Value

Upon successful completion, a value of 0 is returned. If the **sigvec** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **sigvec** system call fails and no new signal handler is installed if one of the following occurs:

- | | |
|---------------|---|
| EFAULT | The <i>invec</i> or <i>outvec</i> parameter points to a location outside of the process's allocated address space |
| EINVAL | The <i>sig</i> parameter is not a valid signal number. |
| EINVAL | An attempt was made to ignore or supply a handler for SIGKILL . |

Related Information

In this book: “kill” on page 2-58, “ptrace” on page 2-100, “sigblock” on page 2-137, “sigpause” on page 2-144, “sigstack” on page 2-148, “sigsetmask” on page 2-146, “sigvec” on page 2-150, and “setjmp, longjmp” on page 3-287.

The **kill** command in *AIX Operating System Commands Reference*.

stat, fstat

Purpose

Gets the status of a file.

Syntax

```
#include <sys/stat.h>
```

```
int stat (path, buf)
char *path;
struct stat *buf;
```

```
int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

Description

The **stat** system call obtains information about the file pointed to by the *path* parameter. Read, write, or execute permission of the named file is not required, but all directories in the path leading to the file must be searchable. The **stat** system call places the information obtained into a structure pointed to by the *buf* parameter.

Use the **fstat** system call to obtain information about an open file pointed to by the *fildes* parameter. The *fildes* parameter is a file descriptor obtained from a successful **open**, **creat**, **dup**, **fcntl**, or **pipe** system call. The **fstat** system call places the information obtained into a structure pointed to by the *buf* parameter.

The **stat** structure pointed to by the *buf* parameter is defined in the **sys/stat.h** header file, and it contains the following members:

```
dev_t    st_dev;    /* ID of the device that contains */
                /* a directory entry for this file */
ino_t    st_ino;    /* Inode number */
ushort   st_mode;    /* File mode; see mknod and chmod */
short    st_nlink;   /* Number of links */
ushort   st_uid;     /* User ID of the file's owner */
ushort   st_gid;     /* Group ID of the file's group */
dev_t    st_rdev;    /* ID of device */
                /* st_rdev is defined only for */
                /* character or block special files */
off_t    st_size;    /* File size in bytes */
time_t   st_atime;   /* Time of last access */
```

stat, fstat

```
time_t st_mtime; /* Time of last data modification */
time_t st_ctime; /* Time of last file status change */
/* Times are measured in seconds since */
/* 00:00:00 GMT, Jan. 1, 1970 */
```

st_atime Time when file data was last accessed. Changed by the following system calls: **creat**, **mknod**, **pipe**, **utime**, and **read**.

st_mtime Time when data was last modified. Changed by the following system calls: **creat**, **mknod**, **pipe**, **utime**, and **write**.

st_ctime Time when file status was last changed. Changed by the following system calls: **chmod**, **chown**, **creat**, **link**, **mknod**, **pipe**, **unlink**, **utime**, and **write**.

Return Value

Upon successful completion, both the **stat** and the **fstat** system calls return a value of 0. If either the **stat** or the **fstat** system calls fail, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **stat** system call fails if one or more of the following are true:

- ENOTDIR** A component of the path prefix is not a directory.
- ENOENT** The named file does not exist.
- EACCES** Search permission is denied for a component of the path prefix.
- EFAULT** The *buf* or *path* parameter points to a location outside of the process's allocated address space.

The **fstat** system call fails if one or more of the following are true:

- EBADF** *fildes* is not a valid open file descriptor.
- EFAULT** The *buf* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “chmod” on page 2-18, “chown” on page 2-21, “creat” on page 2-27, “link” on page 2-60, “mknod” on page 2-67, “pipe” on page 2-93, “read, readx” on page 2-104, “time” on page 2-158, “unlink” on page 2-168, “utime” on page 2-174, “write, writex” on page 2-178, and “stat.h” on page 5-61.

stime

stime

Purpose

Sets the time.

Syntax

```
int stime (tp)
long *tp;
```

Description

The **stime** system call sets the system's time and date. The *tp* parameter points to the time as measured in seconds from 00:00:00 GMT January 1, 1970.

Return Value

Upon successful completion, a value of 0 is returned. If the **stime** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **stime** system call fails if the following is true:

EPERM The effective user ID of the calling process is not superuser.

Related Information

In this book: "time" on page 2-158.

sync

Purpose

Updates the superblock, i-nodes, and delayed blocks.

Syntax

```
void sync ( )
```

Description

The **sync** system call causes all information in memory that should be on disk to be written out. This includes modified superblocks, modified i-nodes, delayed block I/O, and read-write mapped files. The **sync** system call should be used by programs that examine a file system, such as the **df** and **fsck** commands described in *AIX Operating System Commands Reference*.

The writing, although scheduled, is not necessarily complete upon return from the **sync** system call.

Related Information

In this book: “fsync” on page 2-46.

The **sync** command in *AIX Operating System Commands Reference*.

time

time

Purpose

Gets the time.

Purpose

long time ((long *) 0)

long time (*tloc*)
long **tloc*;

Description

The **time** system call returns the current time in seconds since 00:00:00 GMT, January 1, 1970.

If the *tloc* parameter is nonzero, the time is also stored in the location to which the *tloc* parameter points.

Warning: The actions of the **time** system call are undefined if the *tloc* parameter points to a location outside of the process's allocated address space.

Return Value

Upon successful completion, the current time is returned.

Related Information

In this book: "stime" on page 2-156.

times

Purpose

Gets process and child process times.

Syntax

```
#include <sys/types.h>
#include <sys/times.h>

time_t times (buffer)
struct tms *buffer;
```

Description

The **times** system call fills the structure pointed to by the *buffer* parameter with time-accounting information. All time values reported by the **times** system call are in 16ths of a second, unless execution profiling is enabled. When profiling is enabled, **times** reports values in 60ths of a second. (For more information about profiling, see “profil” on page 2-97, “monitor” on page 3-236, and the **cc** and **prof** commands in *AIX Operating System Commands Reference*.)

The **tms** structure is defined in **sys/times.h** and it contains the following members:

```
time_t  tms_utime;
time_t  tms_stime;
time_t  tms_cutime;
time_t  tms_cstime;
```

This information comes from the calling process and each of its terminated child processes for which it has executed a **wait** system call.

tms_utime	The CPU time used while executing instructions in the user space of the calling process.
tms_stime	The CPU time used by the system on behalf of the calling process.
tms_cutime	The sum of the tms_utimes and the tms_cutimes of the child processes.
tms_cstime	The sum of the tms_stimes and the tms_cstimes of the child processes.

Note: The system measures time by counting clock interrupts. The precision of the values reported by the **times** system call depends on the rate at which the clock interrupts occur.

times

Return Value

Upon successful completion, the **times** system call returns the elapsed real time, in 60ths of a second, since an arbitrary reference time in the past (for example, system start-up time). This reference time does not change from one call of **times** to another. If the **times** system call fails, a -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **times** system call fails if the following is true:

EFAULT The *buffer* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “fork” on page 2-44, “profil” on page 2-97, “time” on page 2-158, “wait” on page 2-176, and “monitor” on page 3-236.

The **cc** and **prof** commands in *AIX Operating System Commands Reference*.

ulimit

Purpose

Sets and gets user limits.

Syntax

```
#include <sys/types.h>
```

```
off_t ulimit (cmd, newlimit)  
int cmd;  
off_t newlimit;
```

Description

The **ulimit** system call controls process limits. The *cmd* parameter values are:

- 1 Returns the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.
- 2 Sets the process's file size limit to the value of the *newlimit* parameter. Any process can decrease this limit, but only a process with an effective user ID of superuser can increase the limit.
- 3 Returns the maximum possible break value (see "brk, sbrk" on page 2-14).
- 1004 Sets the maximum possible break value (see "brk, sbrk" on page 2-14). Returns the new maximum break value, which is *newlimit* rounded *up* to the nearest page boundary.
- 1005 Returns the lowest valid stack address. (Note that stacks grow from high addresses to low addresses.)
- 1006 Sets the lowest valid stack address. Returns the new minimum valid stack address, which is *newlimit* rounded *down* to the nearest page boundary.

ulimit

Return Value

Upon successful completion, a nonnegative value is returned. If the **ulimit** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **ulimit** system call fails and the limit remains unchanged if:

- | | |
|---------------|--|
| EPERM | A process with an effective user ID other than superuser attempts to increase the file size limit. |
| EINVAL | The <i>cmd</i> parameter is a value other than 1, 2, or 3. |

Example

To increase the size of the stack segment by 2048 bytes, and set *rc* to the new lowest valid stack address:

```
rc = ulimit(1006, ulimit(1005, 0) - 2048);
```

Related Information

In this book: “brk, sbrk” on page 2-14 and “write, writex” on page 2-178.

umask

Purpose

Sets and gets the value of the file creation mask.

Syntax

```
int umask (cmask)  
int cmask;
```

Description

The **umask** system call sets the process's file mode creation mask to the value of the *cmask* parameter. Only the low-order 9 bits of the *cmask* parameter and the file mode creation mask are used.

Return Value

Upon successful completion, the previous value of the file mode creation mask is returned.

Related Information

In this book: “chmod” on page 2-18, “creat” on page 2-27, “mknod” on page 2-67, “open” on page 2-88, and “stat.h” on page 5-61.

The **sh** and **umask** commands in *AIX Operating System Commands Reference*.

umount

umount

Purpose

Unmounts a file system.

Syntax

```
int umount (dev)  
char *dev;
```

Description

The **umount** system call unmounts a previously mounted file system contained on the block device (also called a *special file*) identified by the *dev* parameter. The *dev* parameter is a pointer to a path name.

After the file system is unmounted, the directory upon which the file system was mounted reverts to its ordinary interpretation as a directory.

The **umount** system call can be invoked only by a process whose effective user ID is superuser.

Return Value

Upon successful completion, a value of 0 is returned. If the **umount** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **umount** system call fails if one or more of the following are true:

- | | |
|----------------|---|
| EPERM | The process's effective user ID is not superuser. |
| ENOENT | <i>dev</i> does not exist. |
| ENOTBLK | <i>dev</i> is not the name of a block special file. |
| EINVAL | <i>dev</i> is not mounted. |
| EBUSY | A file on the device specified by the <i>dev</i> parameter is currently in use. |

- EFAULT** The *dev* parameter points to a location outside of the process's allocated address space.
- ENXIO** *dev* is not currently configured.

Related Information

In this book: "mount" on page 2-69.

The **mount** and **umount** commands in *AIX Operating System Commands Reference*.

uname, unamex

uname, unamex

Purpose

Gets the name of the current AIX system.

Syntax

```
#include <sys/utsname.h>
```

```
int uname (name)  
struct utsname *name;
```

```
int unamex (name)  
struct xutsname *name;
```

Description

The **uname** system call stores information identifying the current system in the structure pointed to by the *name* parameter.

The **uname** system call uses the **utsname** structure, which is defined in the **sys/utsname.h** file, and it contains the following members:

```
char sysname[9];  
char nodename[9];  
char release[9];  
char version[9];  
char machine[9];
```

The **uname** system call returns a null-terminated character string naming the current system in the character array **sysname**. The **nodename** array contains the name that the system is known by on a communications network. The **release** and **version** arrays further identify the system.

The **machine** array identifies the CPU hardware being used. This array contains an eight-character string followed by a terminating null character. The first two characters identify the hardware model. The hardware model identification may be one of the following:

- 10 IBM RT PC Model 10
- 20 IBM RT PC Model 20 or Model 25

The remaining six characters of the **machine** string specify the unique serial number of the machine. Each digit of the serial number is in the range '0' to '9' or 'A' to 'F'.

The **unamex** system call uses the **xutsname** structure, which is defined in the **sys/utsname.h** file, and it contains the following members:

```
    unsigned long  nid;  
    long          reserved[3];
```

For local area networks in which a binary node name is appropriate, *nid* contains such a name.

Return Value

Upon successful completion, a nonnegative value is returned. If the **uname** or **unamex** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **uname** and **unamex** system calls fail if:

EFAULT The *name* parameter points to a location outside of the process's allocated address space.

Related Information

The **uname** command in *AIX Operating System Commands Reference*.

unlink

unlink

Purpose

Removes a directory entry.

Syntax

```
int unlink (path)  
char *path;
```

Description

The **unlink** system call removes the directory entry specified by the *path* parameter.

When all links to a file are removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file are closed.

Return Value

Upon successful completion, a value of 0 is returned. If the **unlink** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **unlink** system call fails and the named file is not unlinked if one or more of the following are true:

- | | |
|----------------|--|
| ENOTDIR | A component of the path prefix is not a directory. |
| ENOENT | The named file does not exist. |
| EACCES | Search permission is denied for a component of the path prefix. |
| EACCES | Write permission is denied on the directory containing the link to be removed. |
| EPERM | The named file is a directory and the effective user ID of the process is not superuser. |
| EBUSY | The entry to be unlinked is the mount point for a mounted file system. |

- ETXTBSY** The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.
- EROFS** The entry to be unlinked is part of a read-only file system.
- EFAULT** The *path* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “close” on page 2-25, “link” on page 2-60, and “open” on page 2-88.

The **rm** command in *AIX Operating System Commands Reference*.

usrinfo

Purpose

Gets and sets user information about the owner of the calling process.

Syntax

```
#include <uinfo.h>

int usrinfo (cmd, buf, count)
int cmd;
char *buf;
int count;
```

Description

The **usrinfo** system call gets and sets information about the owner of the current process. The information is a sequence of null-terminated *name=value* strings. The last string in the sequence is terminated by two successive null characters. A child process inherits the user information of its parent.

The *buf* parameter is a pointer to a user buffer. This buffer is usually **UINFOSIZ** bytes long.

The *count* parameter is the number of bytes of user information to be copied from or to the user buffer.

If the *cmd* parameter is one of the following constants:

- GETUINFO** Copies up to *count* bytes of user information into the buffer pointed to by the *buf* parameter.
- SETUINFO** Sets the user information for the process to the first *count* bytes in the buffer pointed to by the *buf* parameter. The effective user ID of the calling process must be superuser to set the user information.

The user information should at minimum consist of three strings that are typically set by the **login** program. These three strings are:

```
NAME = username
UID = userid
TTY = ttyname
```

If the process has no terminal, *ttyname* should be null.

Return Value

Upon successful completion, a nonnegative integer giving the number of bytes transferred is returned. If the **usrinfo** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **usrinfo** system call fails if one or more of the following are true:

- | | |
|---------------|---|
| EPERM | The <i>cmd</i> parameter is set to SETUINFO and the effective user ID of the process is not superuser. |
| EINVAL | The <i>cmd</i> parameter is not set to SETUINFO or GETUINFO . |
| EINVAL | The <i>cmd</i> parameter is set to SETUINFO and the count parameter is larger than UINFOSIZ . |
| EINVAL | The <i>cmd</i> parameter is SETUINFO and <i>buf</i> does not contain a NAME = entry. |
| EFAULT | The <i>buf</i> parameter points to a location outside of the process's allocated address space. |

Related Information

In this book: “getuinfo” on page 3-207.

The **login** program in *AIX Operating System Commands Reference*.

ustat

ustat

Purpose

Gets file system statistics.

Syntax

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev, buf)
dev_t dev;
struct ustat *buf;
```

Description

The **ustat** system call gets information about the mounted file system identified by device number *dev*. The *dev* parameter is the ID of the device, and it corresponds to the **st_rdev** member of the structure returned by the **stat** system call. See “stat, fstat” on page 2-153 and “stat.h” on page 5-61 for more information about the device ID. The information is stored into a structure pointed to by the *buf* parameter.

The **ustat** structure pointed to by the *buf* parameter is defined in the **ustat.h** file, and it contains the following members:

daddr_t	f_tfree;	/* Total free blocks */
ino_t	f_tinode;	/* Number of free i-nodes */
char	f_fname[6];	/* File system name */
char	f_fpack[6];	/* File system pack name */

Return Value

Upon successful completion, a value of 0 is returned. If the **ustat** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The `ustat` system call fails if one or more of the following are true:

- EINVAL** *dev* is not the device number of a device containing a mounted file system.
- EFAULT** The *buf* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: “`stat`, `fstat`” on page 2-153, and “`fs`” on page 4-72.

utime

utime

Purpose

Sets file access and modification times.

Syntax

```
#include <unistd.h>

int utime (path, times)
char *path;
struct utimbuf *times;
```

Description

The **utime** system call sets the access and modification times of the file pointed to by the *path* parameter to the value of the *times* parameter.

If the *times* parameter is **NULL**, the access and modification times of the file are set to the current time. The effective user ID of the process must be the same as the owner of the file or must have write permission in order to use the **utime** system call in this manner.

If the *times* parameter is not **NULL**, it is a pointer to a **utimbuf** structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or superuser can use the **utime** system call this way.

The **utimbuf** structure pointed to by the *times* parameter is defined in the **unistd.h**, and it contains the following members.

```
time_t actime;    /* Date and time of last access */
time_t modtime;   /* Date and time of last modification */
```

The times in this structure are measured in seconds since 00:00:00 GMT, January 1, 1970.

Return Value

Upon successful completion, a value of 0 is returned. If the **utime** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **utime** system call fails if one or more of the following are true:

- | | |
|----------------|--|
| ENOENT | The named file does not exist. |
| ENOTDIR | A component of the path prefix is not a directory. |
| EACCES | Search permission is denied by a component of the path prefix. |
| EPERM | The effective user ID is not superuser or the owner of the file and the <i>times</i> parameter is not NULL . |
| EACCES | The effective user ID is not superuser or the owner of the file, the <i>times</i> parameter is NULL , and write access is denied. |
| EROFS | The file system containing the file is mounted read-only. |
| EFAULT | The <i>times</i> or <i>path</i> parameter points to a location outside of the process's allocated address space. |

Related Information

In this book: “stat, fstat” on page 2-153.

wait

wait

Purpose

Waits for a child process to stop or terminate.

Syntax

```
int wait (stat_loc)  
int *stat_loc;
```

```
int wait ((int *) 0)
```

Description

The **wait** system call suspends the calling process until it receives a signal that is to be caught, or until any one of the calling process's child processes stops in a trace mode or terminates. **wait** returns without waiting if a child process that hasn't been waited for has already stopped or terminated prior to the call.

If the *stat_loc* parameter is nonzero, 16 bits of information called *status* are stored in the low-order 16 bits of the location pointed to by *stat_loc*. The *status* information can be used to differentiate between stopped and terminated child processes and, if the child process terminated, the *status* information identifies the cause of termination and passes information to the parent process. This is accomplished in the following manner:

- If the child process stopped in a trace mode, then the high-order 8 bits of *status* contain the number of the signal that caused the process to stop and the low-order 8 bits are set equal to 0177 (0x7F).
- If the child process terminated due to an **exit** system call, the low-order 8 bits of *status* are 0 and the high-order 8 bits contain the low-order 8 bits of the parameter that the child passed to the **exit** system call.
- If the child process terminated due to a signal, the high-order 8 bits of *status* are 0 and the low-order 8 bits contain the number of the signal that caused the termination. In addition, if the low-order seventh bit (bit 0200 or 0x80) is set, then a memory image file is produced before **wait** returns.

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes.

Note: The effect of the **wait** system call can be modified by the setting of the **SIGCLD** signal. See "signal" on page 2-139 for details.

Warning: The actions of the **wait** system call are undefined if the *stat_loc* parameter points to a location outside of the process's allocated address space.

Return Value

If the **wait** system call returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. If the **wait** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **wait** system call fails and returns without waiting if one or more of the following are true:

- | | |
|---------------|---|
| ECHILD | The calling process has no existing unwaited-for child processes. |
| EINTR | The wait system call received a signal. |

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “exit, _exit” on page 2-38, “fork” on page 2-44, “pause” on page 2-92, “ptrace” on page 2-100, and “signal” on page 2-139.

write, writex

Purpose

Writes to a file.

Syntax

```
int write (fildes, buf, nbytes)  
int fildes;  
char *buf;  
unsigned int nbytes;
```

```
int writex (fildes, buf, nbytes, ext)  
int fildes;  
char *buf;  
unsigned int nbytes;  
int ext;
```

Description

The **write** system call writes the number of bytes specified by the *nbytes* parameter from the buffer specified by the *buf* parameter to the file associated with the *fildes* parameter.

The *fildes* parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from the **write** system call, the file pointer increments by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

When the **O_APPEND** flag of the file status is set, the file pointer is set to the end of the file prior to each write.

If the **write** system call requests that more bytes be written than there is room for, only as many bytes as there is room for are written and the **write** system call returns an integer equal to the number of bytes written. The next attempt to write nonzero number of bytes will fail (except as noted following). The limit reached can be either the ulimit (see "ulimit" on page 2-161) or the end of the physical medium. A partial write is not permitted for the following:

- If the file being written is a pipe (or FIFO) and the **O_NDELAY** flag of the file flag word is set, then a **write** to a full pipe (or FIFO) returns a count of 0.
- If the file being written is a pipe (or FIFO) and the **O_NDELAY** flag of the file flag word is not set, then a **write** to a full pipe (or FIFO) blocks until space becomes available.

If the file to be written supports enforcement mode record locks and all or part of the region to be written is currently locked by another process, then the action taken depends on the setting of the **O_NDELAY** flag:

- If **O_NDELAY** is set, then **write** returns -1 and sets **errno** to **EAGAIN**.
- If **O_NDELAY** is not set, then the calling process blocks until the lock is released.

For more information about record locks, see “lockf” on page 2-62.

If the file has been mapped, the **write** system call writes to a mapped file segment. If the *filides* file descriptor was used to map the file copy-on-write, then the copy-on-write segment is used. Otherwise, the **write** system call writes to the read-write mapped segment for the file.

Warning: If a process issues a **write** system call to a file that it has not mapped, but that other processes have mapped copy-on-write, then the results are unpredictable. However, if the process first attaches the mapped file in copy-on-write mode with the **shmat** system call, then the **write** to the file is properly reflected in the copy-on-write shared segment.

The **writex** system call performs the same function as **write**, except that it provides communication with character device drivers that require more information or return more status than **write** can handle.

For files, directories, or special files with drivers that do not handle extended operations, the **writex** system call does exactly what the **write** system call does, and the *ext* parameter is ignored.

Each driver interprets the *ext* parameter in a device-dependent way, either as a value or as a pointer to a communication area. The nonextended **write** system call is equivalent to the extended **writex** system call with an *ext* parameter value of 0. Drivers must apply reasonable defaults when the *ext* parameter value is 0.

Return Value

Upon successful completion, the number of bytes actually written is returned. If the **write** or **writex** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **write** and **writex** system calls fail and the file pointer remains unchanged if one or more of the following are true:

- | | |
|---------------|--|
| EBADF | The <i>filides</i> parameter is not a valid file descriptor open for writing. |
| EAGAIN | An enforcement mode record lock is outstanding in the portion of the file that is to be written. |

write, writex

EPIPE	An attempt is made to write to a pipe that is not open for reading by any process. A SIGPIPE signal is also sent to the calling process.
EFBIG	An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see "ulimit" on page 2-161).
EFAULT	The <i>buf</i> parameter points to a location outside of the process's allocated address space.
EDEADLK	A deadlock would occur if the calling process were to sleep until the region to be written was unlocked.
EINTR	A signal was caught during the write system call.

Related Information

In this book: "creat" on page 2-27, "dup" on page 2-30, "lockf" on page 2-62, "lseek" on page 2-65, "open" on page 2-88, "pipe" on page 2-93, and "ulimit" on page 2-161.

Chapter 3. Subroutines

About This Chapter

This chapter gives detailed information about the subroutines (also called *functions*) that are available in standard AIX subroutine libraries. For an explanation of the differences between system calls and subroutines, see the introduction to Chapter 2 on page 2-2. For an explanation of the “Syntax” section of each entry, see “Syntax” on page v. For an explanation of header files, see “Header Files” on page vi.

Each subroutine entry contains a “Library” section that indicates the library where the subroutine is stored. Subroutines are stored in libraries to conserve storage space and to make the program linkage process more efficient. A *library* (sometimes called an *archive*) is a data file that contains copies of a number of individual files and control information that allows them to be accessed individually. See “ar” on page 4-18 and the **ar** command in *AIX Operating System Commands Reference* for more information about libraries.

The libraries that contain the subroutines described in this book are located in the **/usr/lib** directory. By convention, all of them have names of the form **libname.a**, where *name* identifies the specific library.

You do not need to do anything special to use subroutines from the Standard C Library (**libc.a**) or the Run-time Services Library (**librts.a**). The **cc** command automatically searches these libraries for subroutines that a program needs. However, if you use subroutines from another library, you must tell the compiler to search that library. If your program uses subroutines from the library **libname.a**, compile your program with the flag **-lname**. The following example compiles the program **myprog.c**, which uses subroutines from the **libdbm.a**:

```
cc myprog.c -ldb
```

You can specify more than one **-l** flag, but they must be specified *after* any other flags. See the **cc** command in *AIX Operating System Commands Reference* for details.

The libraries discussed in the book are:

- Curses Library (**libcurses.a**)
- Database Library (**libdbm.a**)
- DOS Services Library (**libdos.a**)
- Extended Curses Library (**libcur.a**)
- Floating-Point Library (**libfp.a**)
- Graphics Libraries (**libplot.a**, **libprint.a**, **lib300.a**, and others)
- Math Library (**libm.a**)
- Object File Access Routine Library (**libld.a**)
- Programmers Workbench Library (**libPW.a**)
- Run-time Services Library (**librts.a**)
- Standard C Library (**libc.a**)
- Standard I/O Package (**libc.a**)
- Usability Services Library (**libpanels.a**).

The Standard I/O Package subroutines are actually contained in the Standard C Library (**libc.a**). These subroutines implement a buffered I/O system on top of the basic I/O provided by the system calls. For more information about these subroutines, see “stdio” on page 3-297.

Note: A few of the subroutines are stored in libraries that may not be included in your system configuration. If the linkage editor (**ld**, which is called by the **cc** command) gives you an error message indicating that it cannot find one of the subroutines, then check Appendix g, “Component Cross Reference.” The subroutines listed there are shipped with programs or licensed programs that must be installed separately.

a64l, l64a

a64l, l64a

Purpose

Converts between long integers and base-64 ASCII strings.

Library

Standard C Library (**libc.a**)

Syntax

long a64l (s)
char *s;

char *l64a (l)
long l;

Description

The **a64l** and **l64a** subroutines maintain numbers stored in base-64 ASCII characters. This is a notation in which long integers are represented by up to 6 characters, each character representing a digit in a base-64 notation.

The following characters are used to represent digits:

.	represents	0.
/	represents	1.
0—9	represent	2—11.
A—Z	represent	12—37.
a—z	represent	38—63.

The **a64l** subroutine takes a pointer to a null-terminated character string containing a value in base-64 representation and returns the corresponding **long** value. If the string pointed to by the **s** parameter contains more than 6 characters, the **a64l** subroutine uses only the first 6.

Conversely, the **l64a** subroutine takes a **long** parameter and returns a pointer to the corresponding base-64 representation. If the **l** parameter is 0, then the **l64a** subroutine returns a pointer to a null string.

The value returned by **l64a** is a pointer into a static buffer, the contents of which are overwritten by each call.

abort

Purpose

Generates an IOT fault to terminate the current process.

Library

Standard C Library (**libc.a**)

Syntax

```
int abort ( )
```

Description

The **abort** subroutine causes a **SIGIOT** signal to be sent to the current process. This usually terminates the process and produces a memory dump.

It is possible for the **abort** subroutine to return control if **SIGIOT** is caught or ignored. In this case, **abort** returns the value returned by the **kill** system call.

If **SIGIOT** is neither caught nor ignored, and if the current directory is writable, then the **abort** subroutine produces a memory dump in a file named **core** in the current directory. The shell then displays the message:

```
abort - core dumped
```

Related Information

In this book: “exit, _exit” on page 2-38, “kill” on page 2-58, and “signal” on page 2-139.

The **adb** command in *AIX Operating System Commands Reference*.

abs

abs

Purpose

Returns the absolute value of an integer.

Library

Standard C Library (**libc.a**)

Syntax

```
int abs (i)
int i;
```

Description

The **abs** subroutine returns the absolute value of its integer operand.

Note: A two's-complement integer can hold a negative number whose absolute value is too large for the integer to hold. When given this largest negative value, the **abs** subroutine returns the same value.

Related Information

In this book: “floor, ceil, fmod, fabs” on page 3-156.

assert

Purpose

Verifies a program assertion.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <assert.h>

void assert (expression)
int expression;
```

Description

The **assert** macro puts diagnostics into a program. If *expression* is false (zero), then **assert** writes the following message on the standard error output and aborts the program:

Assertion failed: *expression*, file *filename*, line *linenum*

In the error message, *filename* is the name of the source file and *linenum* is the source line number of the **assert** statement.

If you compile a program with the preprocessor option **-DNDEBUG**, or with the preprocessor control statement **#define NDEBUG** ahead of the **#include <assert.h>** statement, assertions will not be compiled into the program.

Related Information

In this book: “abort” on page 3-5.

The **c++** command in *AIX Operating System Commands Reference*.

atof

atof

Purpose

Converts an ASCII string to a floating-point number.

Library

Standard C Library (**libc.a**)

Syntax

```
double atof(nptr)  
char *nptr;
```

Description

The **atof** subroutine converts a character string, pointed to by the *nptr* parameter, to a double-precision floating-point number. The first unrecognized character ends the conversion.

The **atof** subroutine recognizes a character string when the characters appear in the following order:

1. An optional string of white-space characters
2. An optional sign
3. A string of digits optionally containing a decimal point
4. An optional **e** or **E** followed by an optionally signed integer.

If the string begins with an unrecognized character, **atof** returns the value 0.

If the correct return value overflows, **atof** returns **HUGE** and sets **errno** to **ERANGE**. On underflow, **atof** returns 0.

Related Information

In this book: “scanf, fscanf, sscanf” on page 3-281.

bessel: j0, j1, jn, y0, y1, yn

Purpose

Computes Bessel functions.

Library

Math Library (**libm.a**)

Syntax

#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n; double x;

Description

The **j0** and **j1** subroutines return Bessel functions of **x** of the first kind, of orders 0 and 1, respectively. **jn** returns the Bessel function of **x** of the first kind of order **n**.

The **y0** and **y1** subroutines return the Bessel functions of **x** of the second kind, of orders 0 and 1, respectively. **yn** returns the Bessel function of **x** of the second kind of order **n**. The value of **x** must be positive.

Non-positive parameters cause **y0**, **y1**, and **yn** to return the value **HUGE** and to set **errno** to **EDOM**. They also write a message indicating DOMAIN error to standard error. You can change these error-handling procedures with the **matherr** subroutine.

bessel: j0, j1, jn, y0, y1, yn

Related Information

In this book: “matherr” on page 3-226.

bsearch

Purpose

Performs a binary search.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <search.h>
```

```
char *bsearch ((char *)key, (char *)base, nel, sizeof (*key), compar)  
unsigned int nel;  
int (*compar) ( );
```

Description

The **bsearch** subroutine is a binary search routine generalized from Donald E. Knuth's *The Art of Computer Programming*, Volume 3, 6.2.1, Algorithm B.* It returns a pointer into a table indicating where a datum is found.

The table must already be sorted in increasing order according to the provided comparison function *compar*. The *key* parameter points to the datum to be sought in the table. The *base* parameter points to the element at the base of the table. The *nel* parameter is the number of elements in the table. The *compar* parameter is a pointer to the comparison function, which is called with two parameters that point to the elements being compared.

The comparison function must compare its parameters and return a value as follows:

- If the first parameter is less than the second parameter, *compar* must return a value less than 0.
- If the first parameter is equal to the second parameter, *compar* must return 0.
- If the first parameter is greater than the second parameter, *compar* must return a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

* Reading, Massachusetts: Addison-Wesley, 1981.

bsearch

The pointers *key* and *base* should be of type pointer-to-element, and cast to type pointer-to-character. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Return Value

If the key is found in the table, the **bsearch** returns a pointer to the element found. If the key cannot be found in the table, then **bsearch** returns the value **NULL**.

Related Information

In this book: “lsearch” on page 3-222, “hsearch, hcreate, hdestroy” on page 3-211, “qsort” on page 3-271, and “tsearch, tdelete, twalk” on page 3-316.

cfgabdds

Purpose

Builds and initializes a Define_Device Structure, and then issues a Define_Device SVC.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <sys/bioca.h>

int cfgabdds (argc, argv, ptr, len)
int argc;
char *argv[ ];
char *ptr;
int len;
```

Description

A customize helper program must issue the **Define_Device** SVC for each VRM device driver. The **cfgabdds** subroutine is provided for use in customize helper programs to simplify the task of building the **Define_Device** structure and issuing the SVC. See *AIX Operating System Programming Tools and Interfaces* for more detailed information about using this subroutine.

The **cfgabdds** subroutine takes the following parameters:

- | | |
|-------------|---|
| <i>argc</i> | The number of elements in the <i>argv</i> array passed to the customize helper by the calling program, which is normally the vrnconfig comand. |
| <i>argv</i> | An array of parameters passed to the customize helper by the calling program, which is normally the vrnconfig command. |
| <i>ptr</i> | A pointer to the initialized structure of device-dependent information for the device being added to the system. If the device requires no device-dependent information, then set this parameter to NULL . |
| <i>len</i> | The length in bytes of the structure pointed to by the <i>ptr</i> parameter. If the device requires no device-dependent information, then set this parameter to 0. |

cfgabdds

The customize helper must pass *argc* and *argv* to **cfgabdds** without modification.

Return Value

Upon successful completion, the **cfgabdds** subroutine returns the value **VRCSUCC**. If **cfgabdds** fails, then one of the following values is returned:

VRCKCORP	A configuration file is not in attribute file format.
VRCKUNXF	An error was returned from the SVC that configures the AIX device driver.
VRCKIOPT	An invalid option was specified from vrconfig .
VRCKNOSP	No storage space available. The malloc subroutine returned a NULL value.
VRCKIARG	One or more parameters are invalid.
VRCKSTNF	kaf-file keyword value not found in the <i>argv</i> passed from vrconfig .
VRCKASNF	kaf-file keyword value not found in the <i>argv</i> passed from vrconfig .
VRCKANOP	Unable to open kaf file.
VRCKYINF	Keyword not found in input file.
VRCKYWNF	Keyword not found in kaf file.
VRCKNONE	No processing required by library routine.
VRCKNOBY	Length of device-dependent information exceeds the limit of the Define_Device Structure. (The total space allowed for entire DDS is 500 bytes.)

Related Information

In this book: Appendix F, "Installing Device Drivers."

AIX Operating System Programming Tools and Interfaces.

cfgadev

Purpose

Adds a device.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg01.h>

int cfgadev (master, system, xstanza, vstanza, dstanza, vflag, cflag)
char *master, *system, *xstanza, *vstanza, *dstanza;
int vflag, cflag;
```

Description

The **cfgadev** subroutine adds information about devices and device drivers to the system configuration.

The *master* parameter points to the full path name of the **master** file. The *system* parameter points to the full path name of the **system** file. These files are usually **/etc/master** and **/etc/system**, respectively.

The *xstanza*, *vstanza*, and *dstanza* parameters point to buffers that contain the text of attribute file stanzas. Any one or two of these parameters can be **NULL** pointers, indicating that a stanza of that type is not to be added, but at least one of them must point to a stanza buffer.

The *xstanza* parameter points to an AIX device driver stanza to be added to the *master* file. If the major device number is missing from the stanza, then the **cfgadev** subroutine generates a new one, beginning with decimal 20.

The *vstanza* parameter points to a VRM device driver stanza to be added to the *master* file. If the IOCN is missing from the stanza, then the **cfgadev** subroutine generates a new one, beginning with decimal 1024.

The *dstanza* parameter points to a device stanza to be added to the *system* file. If the IODN is missing from the stanza, then the **cfgadev** subroutine generates a new one, beginning

with decimal 12000. It also generates a minor device number if only the prefix is supplied or if the value is not unique.

The *vflag* parameter is either 1 (for “yes”) or 0 (for “no”), indicating whether to execute the **vrconfig** command after the device stanza is added. If the *vflag* parameter is 1, then **cfgadev** executes the **vrconfig** command with the **-a stname** flag, where *stname* is the name of the device stanza. The **vrconfig** command then processes this stanza for driver addition and produces a shell procedure. The **cfgadev** subroutine then runs this shell procedure, which creates the special file **/dev/stname**, where *stname* is the name of the device stanza in the *system* file. If the **vrconfig** command returns an error, then all stanzas that were added to the *master* and *system* files are deleted.

The *cflag* parameter is either 1 (for “yes”) or 0 (for “no”), indicating whether to attempt to associate the VRM device driver stanza with one that has been defined previously. If the *cflag* parameter is 1, then the **cfgadev** subroutine gets the VRM device driver stanza associated with the device being added. If this VRM device driver stanza contains a **code** keyword, then **cfgadev** searches through the *master* file for other stanzas that contain the same **code** keyword value and that are associated with a device stanza, in the *system* file, with the same **dtype** value. If another stanza is found and if that stanza is defined before the stanza for the device being added, then the **cfgadev** subroutine replaces the **code** keyword with a **copy** keyword to copy the stanza that was found. If the VRM device driver stanza contains a **copy** keyword, then **cfgadev** verifies that the stanza it is copying is defined before the VRM device driver stanza of the device being added.

If the device stanza pointed to by the *dstanza* parameter contains the **admgr** keyword, then its value specifies the name of the device manager’s stanza in the *system* file. The new device is added to the **vdmgr** keyword value list in the device manager’s stanza.

If the device stanza pointed to by the *dstanza* parameter contains the **specproc** keyword, then the program specified by the value of this keyword is executed to perform any special processing required when adding this device. The value of the **specproc** keyword must be the full path name of an executable file. The following arguments are passed to the program using the **argv** mechanism described in “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32. All of them are passed as character strings.

argv[0]	The full path name of the special-processing program
argv[1]	The full path name of the <i>master</i> file
argv[2]	The full path name of the <i>system</i> file
argv[3]	The name of the device stanza
argv[4]	The character string “a”, indicating addition.

If the special processing program fails, then the device is still added to the system, but additional steps may be required before it can be used.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgadev** subroutine fails, then one of the following values is returned:

CFG_BEMP	The <i>xstanza</i> , <i>vstanza</i> , and <i>dstanza</i> parameters are all NULL pointers.
CFG_BFIC	An input stanza is incomplete, or necessary information is missing.
CFG_BFNA	A failure occurred while adding a stanza to the <i>master</i> or <i>system</i> file.
CFG_BFSM	An input stanza buffer can not be updated because the buffer is too small.
CFG_CFLI	The <i>cflag</i> parameter is 1, but the device stanza pointed to by the <i>dstanza</i> parameter contains nocopy = true . Or, the VRM device driver stanza for the new device contains a copy keyword, but the stanza that it copies is not defined before it in the /etc/master file.
CFG_CLSE	An error was detected while trying to close a file.
CFG_FCOR	The <i>master</i> or <i>system</i> file is set up incorrectly.
CFG_MALF	Memory allocation failed because of insufficient space.
CFG_MAXM	The maximum number of minor device numbers has been reached for the driver associated with the device being added.
CFG_MGRF	A failure occurred while updating the device manager's stanza.
CFG_MPRE	The prefix of the device's minor number is neither b nor c .
CFG_OPNE	An error was detected while trying to open a file.
CFG_SLPF	Special processing failed. The device was added but may require some additional steps before it can be used.
CFGT_VLNG	An IOC value, an IODN value, or a major device number could not be generated to complete an input stanza.
CFG_VCFG	The vrconfig command failed.

File

/etc/specials

Related Information

In this book: “attributes” on page 4-20, “master” on page 4-93, and “system” on page 4-133.

The **vrnconfig** command in *AIX Operating System Commands Reference*.

cfgamni

Purpose

Adds a minidisk.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg02.h>

int cfgamni (sysstanza, fsstanza, use, loc)
char *sysstanza, *fsstanza;
int use, loc;
```

Description

The **cfgamni** subroutine creates minidisks. The necessary steps to accomplish this include adding stanzas to the **/etc/system** and the **/etc/filesystems** files, updating coprocessor stanzas, executing the **vrnconfig** command to define the minidisk, and issuing AIX subroutine calls to make the file system usable. The calling process must have an effective ID of superuser.

The *sysstanza* and *fsstanza* parameters point to buffers that contain the text of attribute file stanzas. The *sysstanza* parameter points to a minidisk stanza that is to be added to the **/etc/system** file. The *fsstanza* parameter points to a minidisk stanza that is to be added to the **/etc/filesystems** file.

The *use* parameter specifies how the minidisk is used. The values allowed for this parameter are:

PARTUNIX	AIX file system partition
PARTCOPR	Coprocessor partition
PARTOTHR	A partition for some other use.

The *loc* parameter specifies the approximate location on the physical disk where the minidisk should reside. The values allowed for this parameter are:

PARTLOCH	At the high end of the fixed disk
PARTLOCM	Near the center of the fixed disk

cfgamni

PARTLOCL Near the beginning of the fixed disk.

After the **cfgamni** subroutine completes successfully, the **/etc/system** file contains a stanza for the new minidisk, and the minidisk has been added to the system.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgamni** subroutine fails, then one of the following values is returned:

CFG_NSID	The calling process's effective user ID is not superuser.
CFG_USZF	The format of a stanza is incorrect.
CFG_MAXP	The maximum number of minidisks are already defined.
CFG_VRMF	The vrconfig command could not define the partition.
CFG_APIIE	One or more parameters are incorrect.
CFG_CFEF	The VRM call to create the partition failed.
CFG_UNRW	An unrecoverable read or write error occurred.
CFG_FOPN	An error occurred while opening a file.

Files

/etc/ddi/cpmgr
/etc/filesystems
/etc/system

Related Information

In this book: “attributes” on page 4-20, “master” on page 4-93, and “system” on page 4-133.

The **vrconfig** command in *AIX Operating System Commands Reference*.

cfgaply

Purpose

Applies configuration information.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg03.h>
```

```
int cfgaply (restart)
int restart;
```

Description

The **cfgaply** subroutine rebuilds the AIX kernel. It runs the **make** command to allow you to rebuild the following files:

- /usr/sys/cf/conf.c
- /usr/sys/cf/conf.o
- /usr/sys/specials
- Kernel library files:
 - /usr/sys/lib0
 - /usr/sys/lib1
 - /usr/sys/lib2

The *restart* parameter indicates whether to restart the system after the subroutine completes. If *restart* is a nonzero value, then the system is restarted after completion.

Before attempting to rebuild the kernel, **cfgaply** creates a backup copy of it named */unixdate.seq*, where *date* is the Julian date the backup was created, and *seq* is a sequence number, starting with 1. Since kernel images take up storage space, it is recommended that only one or two backup kernels be kept on the system at a time.

The **cfgaply** subroutine creates a shell procedure named */usr/sys/specials* that contains the **mknod**, **chown**, and **chmod** commands necessary to create the special files (*/dev* files) needed by the new kernel. The **cfgaply** subroutine does *not* run this shell procedure.

Return Value

If the *restart* parameter is nonzero, then the system is restarted and the **cfgaply** subroutine does not return. If *restart* is 0 and **cfgaply** completes successfully, then it returns the value **CFG-SUCC**. If an AIX program fails that **cfgaply** has executed, the return code from that program is returned. If the **cfgaply** subroutine itself fails, then one of the following values is returned:

CFG-ACCS	A failure occurred while accessing the /unix kernel.
CFG-AOPN	The open system call failed.
CFG-ABCK	A failure occurred while reading the kernel and writing to the backup file.
CFG-ACPF	A failure occurred while reading the rebuilt kernel (/usr/sys/unix.std) and copying it to /unix on the root file system. The previous /unix kernel remains intact.
CFG-AMKF	The make command failed. Error messages from make are redirected to the file /usr/sys/make.out .

Files

/unix
/unixdate.seq
/usr/sys/cf/conf.c
/usr/sys/cf/conf.o
/usr/sys/specials
/usr/sys/lib0
/usr/sys/lib1
/usr/sys/lib2
/usr/sys/make.out
/usr/sys/unix.std

Related Information

In this book: “**config**” on page 6-7.

The **config** and **make** commands in *AIX Operating System Commands Reference*.

cfgcadsz

Purpose

Adds or replaces a stanza in an attribute file.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg04.h>

int cfgcadsz (atfile, stanza, stname, after)
CFG__SFT *atfile;
char *stanza;
char *stname;
char *after;
```

Description

The **cfgcadsz** subroutine adds a new stanza or replaces an existing stanza in an attribute file. (For details about attribute files, see “attributes” on page 4-20.)

The *atfile* parameter points to an open attribute file structure. The *stanza* parameter points to the buffer that contains the stanza to be written. The *stname* parameter points to the name of the stanza to be added to the file.

The *after* parameter points to the name of the stanza after which the new stanza is to be inserted. If this parameter is **NULL**, then the stanza is added to the end of the file.

All information that is repeated in the **default** stanza of the attribute file is removed from the new stanza before it is written to the file.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgcadsz** subroutine fails, then the following value is returned:

- CFG_ECLS** An error occurred while closing a file.
- CFG_EOPN** An error occurred while opening a file.
- CFG_SPCE** Memory allocation failed because of insufficient space.
- CFG_UNIO** An unrecoverable I/O error occurred during processing.

Related Information

In this book: “**cfgadev**” on page 3-15, “**cfgamni**” on page 3-19, “**cfgcclsf**” on page 3-25, “**cfgcdlsz**” on page 3-27, “**cfgcopsf**” on page 3-29, “**cfgcrdsz**” on page 3-31, and “**attributes**” on page 4-20.

cfcclsf

Purpose

Closes an attribute file.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <cfc04.h>
```

```
int cfcclsf (atfile)  
CFG__SFT *atfile;
```

Description

The **cfcclsf** subroutine closes an attribute file. (For details about attribute files, see “attributes” on page 4-20.)

The *atfile* parameter points to an open attribute file structure.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfcclsf** subroutine fails, then the following value is returned:

CFG_UNIO Unrecoverable I/O error occurred during processing.

Related Information

In this book: “cfgcadsz” on page 3-23, “cfgcdlsz” on page 3-27, “cfgcopsf” on page 3-29, “cfgcrdsz” on page 3-31, and “attributes” on page 4-20.

cfgcdlsz

Purpose

Deletes a stanza from an attribute file.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg04.h>

int cfgcdlsz (atfile, stname)
CFG__SFT *atfile;
char *stname;
```

Description

The **cfgcdlsz** subroutine deletes a stanza from an attribute file. (For details about attribute files, see “attributes” on page 4-20.)

The *atfile* parameter points to an open attribute file structure. The *stname* parameter points to the name of the stanza to be deleted from the file.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgcdlsz** subroutine fails, then one of the following values is returned:

- CFG_ECLS** An error occurred while closing a file.
- CFG_EOPN** An error occurred while opening a file.
- CFG_SPCE** Memory allocation failed because of insufficient space.
- CFG_SZBF** The file contains a stanza that is larger than the maximum allowable stanza size.

cfgcdlsz

CFG_SZNF The requested stanza to be deleted was not found in the file.

CFG_UNIO An unrecoverable I/O error occurred during processing.

Related Information

In this book: “cfgddev” on page 3-33, “cfgdmni” on page 3-36, “cfgcadsz” on page 3-23, “cfgcclsf” on page 3-25, “cfgcopsf” on page 3-29, “cfgcrdsz” on page 3-31, and “attributes” on page 4-20.

cfgcopsf

Purpose

Opens an attribute file.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg04.h>
```

```
CFG__SFT *cfgcopsf (path)
char *path;
```

Syntax

The **cfgcopsf** subroutine opens an attribute file for update. (For details about attribute files, see “attributes” on page 4-20.)

The *path* parameter points to the full path name of the file to be opened.

The **cfgcopsf** subroutine calls the **fopen** subroutine to open the file for update. If the call to **fopen** is successful, then **cfgcopsf** allocates a **CFG__SFT** structure. This structure contains the file descriptor returned by **fopen**, a pointer to a default stanza buffer for reads, a pointer to an array of indexes in a default stanza buffer, and the full path name of the file that was opened.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value

Upon successful completion, the **cfgcopsf** subroutine returns a pointer to an open attribute file structure. If the **cfgcopsf** subroutine fails, it returns a **NULL** pointer.

Related Information

In this book: “cfgcadsz” on page 3-23, “cfgccclsf” on page 3-25, “cfgcdlsz” on page 3-27, “cfgcrdsz” on page 3-31, “fopen, freopen, fdopen” on page 3-157, and “attributes” on page 4-20.

cfgcrdsz

Purpose

Reads an attribute file stanza.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg04.h>

int cfgcrdsz (atfile, stanza, nbytes, stname)
CFG__SFT *atfile;
char *stanza;
int nbytes;
char *stname;
```

Description

The **cfgcrdsz** subroutine reads one stanza from an attribute file. A specific stanza may be requested, or the next stanza in the file can be read. When a stanza is read, any information contained in a **default** stanza preceding it in the file will be added to the information returned in the buffer. (For details about attribute files, see “attributes” on page 4-20.)

The *atfile* parameter points to an open attribute file structure.

The *stanza* parameter points to the buffer into which the stanza will be read.

The *nbytes* parameter is the size in bytes of the buffer pointed to by the *stanza* parameter.

The *stname* parameter points to a string containing the name of the stanza to be read. If this parameter is a **NULL** pointer, then the next stanza in the file is read.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgcrdsz** subroutine fails, then one of the following values is returned:

- CFG_EOF** The next stanza was requested, but the end of the file has been reached.
- CFG_SZNF** The requested stanza was not found in the file.
- CFG_SZBF** The requested stanza is longer than *nbytes* bytes.
- CFG_UNIO** Unrecoverable I/O error occurred during processing.

Related Information

In this book: “**cfgcadsz**” on page 3-23, “**cfgccclsf**” on page 3-25, “**cfgcdlsz**” on page 3-27, “**cfgcopsf**” on page 3-29, and “**attributes**” on page 4-20.

cfgddev

Purpose

Deletes a device.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg01.h>

int cfgddev (master, system, dstname, vflag)
char *master, *system, *dstname;
int vflag;
```

Description

The **cfgddev** subroutine deletes information about devices and device drivers from the system configuration.

The *master* parameter points to the full path name of the **master** file. The *system* parameter points to the full path name of the **system** file. These files are usually **/etc/master** and **/etc/system**, respectively. The *dstname* parameter points to a string containing the name of the stanza in the *system* file of the device to be deleted.

The *vflag* parameter is either 1 (for “yes”) or 0 (for “no”). If the *vflag* parameter is 1, then **cfgddev** executes the **vrconfig** command with the **-d dstname** flag. The **vrconfig** command then processes the named stanza for driver deletion and produces a shell procedure. The **cfgddev** subroutine then runs this shell procedure to delete the special file (**/dev** file) for the device. If the **vrconfig** command returns an error, then the device is not deleted.

The **cfgddev** subroutine then gets the VRM device driver stanza associated with the device being deleted. If it contains a **code** keyword, then **cfgddev** searches the *master* file for other VRM device driver stanzas that copy this stanza by specifying its name as the value of a **copy** keyword. If any are found, they are updated so that the first of these stanzas defined in the *master* file contains the **code** keyword, and the other stanzas copy the first stanza.

If the VRM device driver stanza for the device being deleted contains a **copy** keyword, then it is replaced with a **code** keyword whose value is the same as the value of the **code** keyword in the stanza it is copying.

If the device stanza being deleted from the *system* file contains an **admgr** keyword, then its value is the name of the device manager's stanza in the *system* file. The device is deleted from the **vdmgr** keyword value list in the device manager's stanza.

If the device stanza named by the *dstname* parameter contains the **specproc** keyword, then the program specified by the value of this keyword is executed to perform any special processing required when deleting this device. The value of the **specproc** keyword must be the full path name of an executable file. The following arguments are passed to the program using the **argv** mechanism described in "exec: execl, execv, execl, execve, execlp, execvp" on page 2-32. All of them are passed as character strings.

argv[0]	The full path name of the special-processing program
argv[1]	The full path name of the <i>master</i> file
argv[2]	The full path name of the <i>system</i> file
argv[3]	The name of the device stanza
argv[4]	The character string "d", indicating deletion.

If the special processing program fails, then the device is still deleted from the system, but some additional steps may be required to clean up the system.

The device stanza associated with the deleted device is then deleted from the *system* file.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgddev** subroutine fails then one of the following values is returned:

CFG_CLSE	An error was detected while trying to close a file.
CFG_CPYF	A failure occurred while trying to update the VRM driver stanzas that copy the driver stanza of the device being deleted.
CFG_DVND	The device could not be deleted from the <i>system</i> file.
CFG_DVNF	The device to be deleted cannot be found in the <i>system</i> file.
CFG_FCOR	The <i>master</i> or <i>system</i> file is set up incorrectly.
CFG_MALF	Memory allocation failed because of insufficient space.
CFG_MGRF	A failure occurred while updating the device manager's stanza for the device being deleted.

- CFG_OPNE** An error was detected while trying to open a file.
- CFG_SLPF** Special processing failed. The device is deleted but some additional steps may be required to clean up the system.
- CFG_VCFG** The **vrconfig** command failed.

Files

/etc/specials

Related Information

In this book: “attributes” on page 4-20, “master” on page 4-93, and “system” on page 4-133.

The **vrconfig** command in *AIX Operating System Commands Reference*.

cfgdmni

cfgdmni

Purpose

Deletes a minidisk.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg02.h>

int cfgdmni (sysstname, fsstname)
char *sysstname, *fsstname;
```

Description

The **cfgdmni** subroutine deletes a minidisk from the system. The necessary steps to accomplish this include calling the minidisk manager, executing the **vrnconfig** command, removing the minidisk stanzas from the **/etc/system** and **/etc/filesystems** files, and removing the coprocessor stanza references.

The *sysstname* parameter is a pointer to the name of the stanza in the **/etc/system** file that describes the minidisk that is to be deleted.

The *fsstname* parameter is a pointer to the name of the stanza in the **/etc/filesystems** file if the minidisk is an AIX minidisk. If the minidisk is not an AIX minidisk, then the *fsstname* parameter must be **NULL**.

After the **cfgdmni** subroutine completes successfully, the minidisk is deleted from the system and the minidisk stanza has been deleted from the **/etc/system** file, and, if appropriate, from the **/etc/filesystems** file or from the **/etc/ddi/cpmgr** file.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgdmni** subroutine fails, then one of the following is returned:

CFG_NSID	The calling process's effective user ID is not superuser.
CFG_VRMF	The vrmdir command could not delete the partition.
CFG_APIE	One or more parameters are incorrect.
CFG_CFEF	The VRM call to delete the partition failed.
CFG_USZF	The stanza specified by the <i>sysstname</i> parameter could not be found in the <i>/etc/system</i> file.
CFG_UNRW	An unrecoverable read or write error occurred.
CFG_FOPN	An error occurred while opening a file.

Files

/etc/ddi/cpmgr
/etc/filesystems
/etc/system

Related Information

In this book: “attributes” on page 4-20, “master” on page 4-93, and “system” on page 4-133.

The **vrmdir** command in *AIX Operating System Commands Reference*.

clock

clock

Purpose

Reports CPU time used.

Library

Standard C Library (**libc.a**)

Syntax

long clock ()

Description

The **clock** subroutine returns the amount of CPU time (in microseconds) used since the first call to **clock**.

The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed a **wait** system call or a **system** subroutine. The nominal resolution of the clock is 16.667 milliseconds if the process is being profiled; otherwise, it is 100 milliseconds. See “monitor” on page 3-236 and “profil” on page 2-97 for information about profiling a process.

Note: The value returned by the **clock** subroutine is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned wraps around after accumulating approximately 2147 seconds of CPU time (about 36 minutes).

Related Information

In this book: “times” on page 2-159, “wait” on page 2-176, and “system” on page 3-305.

crypt, encrypt

Purpose

Encrypts user passwords.

Library

Standard C Library (**libc.a**)

Syntax

```
char *crypt (key, salt)  
char *key, *salt;
```

```
void encrypt (block)  
char *block;
```

Description

The **crypt** and **encrypt** subroutines encrypt user passwords. They are based on a one-way hashing encryption algorithm with variations intended to frustrate the use of hardware-implemented key searches. These subroutines are provided for compatibility with UNIX system implementations, and no assertion is made about the strength of the algorithm.

The *key* parameter is a user's typed password. The *salt* parameter is a two-character string chosen from the set [a-zA-Z0-9./].

The *salt* parameter is used to perturb the hashing algorithm in one of 4096 different ways, after which the password is used as the key to repeatedly encrypt a constant string. The return value points to the encrypted password. The first two characters of the return value are the string entered in the *salt* parameter.

The **crypt** subroutine uses a character array of length 64 containing only the values (char) 0 and (char) 1. This string is divided into groups of eight characters each, and the low-order bit in each group is ignored. This provides a 56-bit key, which is set into the machine by **crypt**.

The **encrypt** subroutine provides somewhat primitive access to the actual hashing algorithm. The *block* parameter is a 64-character array containing only the values (char) 0 and (char) 1. **encrypt** modifies this array in place, producing a similar array that has been subjected to the hashing algorithm using the key set by **crypt**.

crypt, encrypt

Return Value

The **crypt** subroutine returns a pointer to the encrypted password. The first two characters of it are the same as the *salt* parameter.

Note: The return value points to static data that is overwritten by subsequent calls.

Related Information

In this book: “getpass” on page 3-201, and “passwd” on page 4-107.

The **login** and **passwd** commands in *AIX Operating System Commands Reference*.

ctermid

Purpose

Generates a file name for terminal.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
char *ctermid (s)  
char *s;
```

Description

The **ctermid** subroutine generates the path name of the controlling terminal for the current process and stores it in a string.

If the *s* parameter is a **NULL** pointer, the string is stored in an internal static area and the address is returned. The next call to **ctermid** overwrites the contents of the internal static area.

If the *s* parameter is not a **NULL** pointer, it points to a character array of at least **L_ctermid** elements as defined in the **stdio.h** header file. The path name is placed in this array and the value of *s* is returned.

The difference between the **ctermid** and **ttyname** subroutines is that **ttyname** must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while **ctermid** returns a string (**/dev/tty**) that refers to the terminal if used as a file name. Thus **ttyname** is useful only if the process already has at least one file open to a terminal.

Related Information

In this book: “ttyname, isatty” on page 3-318.

ctime, localtime, gmtime, asctime, tzset

Purpose

Converts date and time to string representation.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <time.h>
```

```
char *ctime (clock)
long *clock;
```

```
struct tm *localtime (clock)
long *clock;
```

```
struct tm *gmtime (clock)
long *clock;
```

```
char *asctime (tm)
struct tm *tm;
```

```
void tzset ( )
```

```
extern long timezone;
extern int daylight;
extern char *tzname[2];
```

Description

The **ctime** subroutine converts a time value pointed to by the *clock* parameter, representing the time in seconds since 00:00:00 Greenwich Mean Time (GMT), January 1, 1970, into an ASCII 26-character string in the following form:

```
Sun Sep 16 01:03:52 1973\n\n0
```

The width of all fields is constant.

The **localtime** subroutine converts the long integer pointed to by the *clock* parameter into the time in seconds since 00:00:00 GMT, January 1, 1970. **localtime** corrects for the time zone and for Daylight Savings Time, if it is in effect.

The **gmtime** subroutine converts the long integer pointed to by the *clock* parameter directly into Greenwich Mean Time, which is the time the AIX system uses.

Both **localtime** and **gmtime** return pointers to **tm** structures. The **tm** structure is defined in the **time.h** header file, and it contains the following members:

ctime, localtime, gmtime, asctime, tzset

```
int tm_sec;    /* Seconds (0 - 59) */
int tm_min;    /* Minutes (0 - 59) */
int tm_hour;   /* Hours (0 - 23) */
int tm_mday;   /* Day of month (1 - 31) */
int tm_mon;    /* Month of year (0 - 11) */
int tm_year;   /* Year - 1900 */
int tm_wday;   /* Day of week (Sunday = 0) */
int tm_yday;   /* Day of year (0 - 365) */
int tm_isdst;  /* Nonzero = Daylight savings time */
```

The **asctime** subroutine converts a **tm** structure to a 26-character string in the same format as **ctime**.

Declarations of all the subroutines, externals, and the **tm** structure, are in the **time.h** header file.

The external variable **timezone** contains the difference, in seconds, between GMT and local standard time. For example, **timezone** is $5 \times 60 \times 60$ for Eastern Standard Time.

The external variable **daylight** is nonzero only if the standard U.S.A. Daylight Savings Time conversion should be applied. The conversion program adjusts for the peculiarities of this conversion in 1974 and 1975. If necessary, a table for these years can be extended.

If the environment variable **TZ** is present, **asctime** uses its value to override the default time zone. The value of the **TZ** variable is a three-letter time zone name, followed by a number representing the difference between local time and Greenwich Mean Time in hours, followed by an optional three-letter name for a daylight savings time zone. For example, the setting for New Jersey would be **EST5EDT**.

Setting the environment variable **TZ** changes the values of the external variables **timezone** and **daylight**. In addition, the time zone names contained in the external variable **tzname** are set from **TZ**:

```
char *tzname[2] = {"EST", "EDT"};
```

The **tzset** subroutine sets these external variables from **TZ**. **tzset** is called by **asctime** and can also be called explicitly by the user.

TZ is usually set by default to a value in the **/etc/environment** file when the system is started up. (For information about this file, see "environment" on page 5-44.)

Note: The return values point to static data that is overwritten by each call.

Related Information

In this book: “time” on page 2-158, “getenv” on page 3-193, “profile” on page 4-121, and “environment” on page 5-44.

ctype

ctype

Purpose

Classifies characters.

Library

Standard C Library (**libc.a**)

Syntax

#include <ctype.h>

int isalpha (c)
int c;

int isspace (c)
int c;

int isupper (c)
int c;

int ispunct (c)
int c;

int islower (c)
int c;

int isprint (c)
int c;

int isdigit (c)
int c;

int isgraph (c)
int c;

int isxdigit (c)
int c;

int iscntrl (c)
int c;

int isalnum (c)
int c;

int isascii (c)
int c;

Description

The **ctype** macros classify character-coded integer values by table look-up. Each of these macros returns a nonzero value for “true” and 0 for “false.”

The **isascii** macro is defined for all integer values. The other macros return a meaningful value only if **isascii** returns “true” for the same *c* value, or if *c* is **EOF**. (See “stdio” on page 3-297 for information about the value **EOF**.)

The following list shows the set of values for which each macro returns a nonzero (“true”) value:

isalpha	<i>c</i> is a letter.
isupper	<i>c</i> is an uppercase letter.
islower	<i>c</i> is a lowercase letter.
isdigit	<i>c</i> is a digit in the range [0-9].
isxdigit	<i>c</i> is a hexadecimal digit in the range [0-9], [A-F] or [a-f].
isalnum	<i>c</i> is alphanumeric (a letter or a digit).
isspace	<i>c</i> is a space, tab, carriage return, new-line, vertical tab, or form-feed character.
ispunct	<i>c</i> is a punctuation character (neither a control character nor alphanumeric).
isprint	<i>c</i> is a printing character, ASCII space (040 or 0x20) through ~ (0176 or 0x7E).
isgraph	<i>c</i> is a printing character, like isprint but, unlike isprint , isgraph returns false (0) for the space character.
isctrl	<i>c</i> is a delete character (0177 or 0xFF) or an ordinary control character (less than 040 or 0x20).
isascii	<i>c</i> is an ASCII character whose value is less than 0200 (0x80).

Related Information

In this book: “ascii” on page 5-3.

curses

curses

Purpose

Controls cursor movement and windowing.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

Description

Note: The **curses** package of subroutines is included here only for compatibility with existing programs. For information about the enhanced screen-handling subroutine library, see "Extended Curses" on page 3-123.

The **curses** subroutine package updates the screen with reasonable optimization.

In order to initialize the routines, the routine **initscr** must be called before any of the other routines that deal with windows and screens are used. The routine **endwin** should be called before exiting. To get character-at-a-time input without echoing, call the **nonl**, **cbreak**, and **noecho** routines. Most interactive, screen-oriented programs require the character-at-a-time input without echoing.

The full **curses** interface permits manipulation of data structures called **windows**, which can be thought of as two-dimensional arrays of characters representing all or part of a screen. A default window called **stdscr** is supplied, and others can be created with the **newwin** routine. Windows are referred to by variables declared **WINDOW ***. The type **WINDOW** is defined in **curses.h** to be a C structure. These data structures are manipulated with the routines described following, among which the most basic are **move** and **addch**. Then the **refresh** routine is called, telling the routines to make the screen look like **stdscr**. More general versions of these routines are included with names beginning with **w** allowing you to specify a window. The routines not beginning with a **w** affect **stdscr**.

Minicurses is a subset of curses that does not allow manipulation of more than one window. To invoke this subset, use **-DMINICURSES** as a cc option. This level is smaller and faster than the full curses.

If the environment variable **TERMINFO** is defined, any program using curses is checked for a local terminal definition before checking in the standard place. For example, if the standard place is **/usr/lib/terminfo**, and **TERM** is set to **vt100**, then normally, the compiled file is found in **/usr/lib/terminfo/v/vt100**. (The directory name **v** is copied from the first letter of **vt100** to avoid creating huge directories.) However, if **TERMINFO** is set to **/usr/mark/myterms**, curses first checks **/usr/mark/myterms/v/vt100**. If this fails, curses then checks **/usr/lib/terminfo/v/vt100**. This is useful for developing experimental definitions or when write permission in **/usr/lib/terminfo** is not available.

Note: The plotting library, **plot** and the curses library, **curses** both use the names **erase** and **move**. The **curses** versions are macros. If you need both libraries, put the **plot** code in a different source file than the **curses** code, or include the following statements in the **plot** code:

```
#undef move()
#undef erase()
```

Routines

The routines listed here can be called when using the full curses. Those marked with an asterisk can be called when using minicurses.

addch(*ch*)*

Add a character to **stdscr** (like **putchar**), wrapping to the next line at the end of a line.

addstr(*str*)*

Call **addch** with each character in *str*

attroff(*attrs*)*

Turn off the attributes named in *attrs*

attron(*attrs*)*

Turn on the attributes named in *attrs*

attrset(*attrs*)*

Set current attributes to those specified in *attrs*

baudrate ()*

Current terminal speed

beep ()*

Sound beep on terminal

box(*win*, *vert*, *hor*)

Draw a box around edges of *win*. The *vert* and *hor* parameters are the characters to use for vertical and horizontal edges of the box.

cbreak ()*

Set cbreak mode

clear ()

Clear **stdscr**

clearok(*win*, *bf*)

Clear screen before next redraw of *win*

clrtoBot ()

Clear to bottom of **stdscr**

clrtoeol ()

Clear to end of line on **stdscr**

delay_output(*ms*)*

Insert *ms* millisecond pause in output

delch ()

Delete a character

deleteln ()

Delete a line

delwin(*win*)

Delete window *win*

doupdate ()

Update screen from all wnooutrefresh

echo ()*

Set echo mode

endwin ()*

End window modes

erase ()

Erase **stdscr**

erasechar ()

Return user's erase character

fixterm ()

Restore terminal to *in curses* state

flash ()

Flash screen or beep

flushinp ()*

Throw away any type-ahead

getch ()*

Get a character from **tty**

getstr(str)

Get a string through **stdscr**

gettmode ()

Establish current **tty** modes

getyx(win, y, x)

Get (y, x) coordinates

has_ic ()

True if terminal can do insert character

has_il ()

True if terminal can do insert line

idlok(win, bf)*

Use terminal's insert/delete line if *bf*!=0

inch ()

Get character at current (y, x) coordinates

initscr ()*

Initialize screens

insch(c)

Insert a character

insertln ()

Insert a line

curses

intrflush(*win*, *bf*)

Interrupts flush output if *bf* is true

keypad(*win*, *bf*)

Enable keypad input

killchar ()

Return current user's **kill** character

leaveok(*win*, *flag*)

OK to leave cursor anywhere after refresh if *flag*!=0 for *win*; otherwise cursor must be left at current position

longname ()

Return verbose name of terminal

meta(*win*, *flag*)*

Allow metacharacters on input if *flag*!=0

move(*y*, *x*, *ch*)*

Move to (*y*, *x*) on **stdscr**

mvaddch(*y*, *x*, *ch*)

Move (*y*, *x*) then add *ch*

mvaddstr(*y*, *x*, *str*)

Move (*y*, *x*) then add *str*

mvcur(*oldrow*, *oldcol*, *newrow*, *newcol*)

Move cursor from current position to another position

mvdelch(*y*, *x*)

Move (*y*, *x*) then delete a character

mvgetch(*y*, *x*)

Move (*y*, *x*) then get a character from **tty**

mvgetstr(*y*, *x*, *str*)

Move (*y*, *x*) then get a string through **stdscr**

mvinch(*y*, *x*)

Move (*y*, *x*) then get the character at current (*y*, *x*) coordinates

mvinsch(*y, x, c*)

Move (*y, x*) then insert the character *c*

mvprintw(*y, x, fmt, args*)

Move (*y, x*) then get print on **stdscr**

mvscanw(*y, x, fmt, args*)

Move (*y, x*) then scan through **stdscr**

mvwaddch(*win, y, x, ch*)

Move (*y, x*) then add the character *ch* to *win*

mvwaddstr(*win, y, x, str*)

Move (*y, x*) then add the string *str* to *win*

mvwdelch(*win, y, x*)

Move (*y, x*) then delete a character from *win*

mvwgetch(*win, y, x*)

Move (*y, x*) then get a character through *win*

mvwgetstr(*win, y, x, str*)

Move (*y, x*) then get a string through *win*

mvwin(*win, by, bx*)

move *win* so that the upper left-hand corner is located at (*y, x*)

mvwinch(*win, y, x*)

Move (*y, x*) then get the character at current (*y, x*) in *win*

mvwinsch(*win, y, x, c*)

Move (*y, x*) then insert the character *c* into *win*

mvwprintw(*win, y, x, fmt, args*)

Move (*y, x*) then **printf** on **stdscr**

mvwscanw(*win, y, x, fmt, args*)

Move (*y, x*) then **scanf** through **stdscr**

newpad(*nlines, ncols*)

Create a new pad with given dimensions

newterm(*type, fd*)

Set up new terminal of given type to output on *fd*

newwin(*lines, cols, begin-y, begin-x*)

Create a new window

nl ()*

Set newline mapping

nocbreak ()*

Unset cbreak mode

nodelay(*win, bf*)

Enable nodelay input mode through **getch**

noecho ()*

Unset echo mode

nonl ()*

Unset newline mapping

noraw ()*

Unset raw mode

overlay(*win1, win2*)

Overlay *win1* on *win2*

overwrite(*win1, win2*)

Overwrite *win1* on top of *win2*

pnoutrefresh(*pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol*)

Like **prefresh** but with no output until **doupdate** called

prefresh(*pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol*)

Refresh from *pad* starting with given upper left corner of pad with output to given portion of screen

printw(*fmt, arg1, arg2, ...*)

Print on **stdscr**

raw ()*

Set raw mode

refresh ()*

Make current screen look like **stdscr**

resetterm ()*

Set **tty** modes to *out of curses* state

resetty ()*

Reset **tty** flags to stored value

saveterm ()*

Save current modes as *in curses* state

savetty ()*

Store current **tty** flags

scanw(fmt, arg1, arg2, ...)

Scanf through **stdscr**

scroll(win)

Scroll *win* one line

scrollok(win, flag)

Allow terminal to scroll if *flag*!=0

set_term(new)

Now talk to terminal *new*

setscrreg(t, b)

Set user scrolling region to lines *t* through *b*

setterm(type)

Establish terminal with a give type

standend ()*

Clear standout mode attribute

standout ()*

Set standout mode attribute

subwin(win, lines, cols, begin-y, begin-x)

Create a subwindow

touchwin(win)

Change all of *win*

traceoff ()

Turn off debugging trace output

traceon ()

Turn on debugging trace output

typeahead(*fd*)

Check file descriptor *fd* to check type-ahead

unctrl(*ch*)*

Printable version of *ch*

waddch(*win, ch*)

Add the character *ch* to *win*

waddstr(*win, str*)

Add the string *str* to *win*

wattroff(*win, attrs*)

Turn off *attrs* in *win*

wattron(*win, attrs*)

Turn on *attrs* in *win*

wattrset(*win, attrs*)

Set attributes in *win* to *attrs*

wclear(*win*)

Clear *win*

wclrtoobot(*win*)

Clear to bottom of *win*

wclrtoeol(*win*)

Clear to end of line on *win*

wdelch(*win, c*)

Delete the character *c* from *win*

wdeleteln(*win*)

Delete line from *win*

werase(*win*)

Erase *win*

wgetch(*win*)

Get a character through *win*

wgetstr(*win*, *str*)

Get the string *str* through *win*

winch(*win*)

Get the character at current (y, x) in *win*

winsch(*win*, *c*)

Insert the character *c* into *win*

winsertln(*win*)

Insert line into *win*

wmove(*win*, *y*, *x*)

Set current (y, x) coordinates on *win*

wnoutrefresh(*win*)

Refresh but no screen output

wprintw(*win*, *fmt*, *arg1*, *arg2*, . . .)

printf on *win*

wrefresh(*win*)

Make screen look like *win*

wscanw(*win*, *fmt*, *arg1*, *arg2*, . . .)

scanf through *win*

wsetscrreg(*win*, *t*, *b*)

Set scrolling region of *win*

wstandend(*win*)

Clear standout attribute in *win*

wstandout(*win*)

Set standout attribute in *win*

Terminfo Level Routines

These routines should be called by programs that have to deal directly with the **terminfo** database. Due to the low level of this interface, its use is discouraged. Initially, **setupterm** should be called. This defines the set of terminal-dependent variables defined in the **terminfo** file. The include files **curses.h** and **term.h** should be included to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through **tparm** to instantiate them. All **terminfo** strings (including the output of **tparm**) should be printed with **tputs** or **putp**. Before exiting, **resetterm** should be called to restore the **tty** modes. Programs desiring shell escapes or suspending with **Ctrl-Z** can call **resetterm** before the shell is called and **fixterm** after returning from the shell.

fixterm ()

Restore **tty** modes for **terminfo** use (called by **setupterm**)

resetterm ()

Reset **tty** modes to state before program entry

setupterm(term, fd, rc)

Read in database. Terminal type is the character string *term*, all output is to system file descriptor *fd*. A status value is returned in the integer pointed to by *rc*. A status value of 1 is normal. The simplest call would be **setupterm**(0, 1, 0) which uses all defaults.

tparm(str, p1, p2, ..., p9)

Instantiate the string *str* with parms *p*_i.

tputs(str, affcnt, putc)

Apply padding information to string *str*. *affcnt* is the number of lines affected, or 1 if not applicable. *putc* is a **putchar**-like routine to which the characters are passed one at a time.

putp(str)

Calls **tputs**(*str*, 1, **putchar**)

vidputs(attrs, putc)

Output the string to put terminal in video attribute mode *attrs*. Characters are passed to the **putchar**-like routine *putc*.

vidattr(attrs)

Like **vidputs**, but outputs through **putchar**.

Termcap Compatibility Routines

These routines are included for compatibility with programs that require **termcap**. Their parameters are the same as for **termcap**, and they are emulated using the **terminfo** database.

tgetent(*bp, name*)

Looks up **termcap** entry for *name*.

tgetflag(*id*)

Get boolean entry for *id*

tgetnum(*id*)

Gets numeric entry for *id*.

tgetstr(*id, area*)

Gets string entry for *id*. The *area* parameter is ignored when **terminfo** is used.

tgoto(*cap, col, row*)

Applies parameters to given *cap*.

tputs(*cap, affcnt, fn*)

Applies padding to *cap* calling *fn* as **putchar**.

Attributes

The following video attributes can be passed to the routines **attron**, **attroff**, and **attrset**.

A_STANDOUT	The terminal's best highlighting mode
A_UNDERLINE	Underlined
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_INVIS	Invisible (blanked or zero-intensity)
A_PROTECT	Protected
A_ALTCHARSET	Alternate character set
A_NORMAL	Normal attributes

Function Keys

The following function keys might be returned by **getch** if **keypad** has been enabled. Note that not all of these are currently supported due to lack of definitions in **terminfo**, or due to the terminal not transmitting a unique code when the key is pressed.

KEY_BREAK	Break key (unreliable)
KEY_DOWN	Down-arrow key
KEY_UP	Up-arrow key
KEY_LEFT	Left-arrow key
KEY_RIGHT	Right-arrow key
KEY_HOME	Home key
KEY_BACKSPACE	Backspace (unreliable)
KEY_F(n)	Function key F n , where n is an integer from 0 to 63
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert character or enter insert mode
KEY_EIC	Exit insert character mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backwards (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send (unreliable)
KEY_SRESET	Soft (partial) reset (unreliable)
KEY_RESET	Reset or hard reset (unreliable)
KEY_PRINT	Print or copy
KEY_LL	Home down or bottom (lower left)
KEY_A1	Upper left key of keypad
KEY_A3	Upper right key of keypad
KEY_B2	Center key of keypad
KEY_C1	Lower left key of keypad
KEY_C3	Lower right key of keypad

Related Information

In this book: “Extended Curses” on page 3-123, and “terminfo” on page 4-141.

cuserid

cuserid

Purpose

Gets the alphanumeric user name associated with the current process.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
char *cuserid (s)  
char *s;
```

Description

The **cuserid** subroutine generates a character string representing the user name of the owner of the current process.

If the *s* parameter is a **NULL** pointer, then the character string is stored into an internal static area, the address of which is returned.

If the *s* parameter is not a **NULL** pointer, then the character string is stored into the array pointed to by the *s* parameter. This array must contain at least **L_cuserid** characters. **L_cuserid** is a constant defined in the **stdio.h** header file.

If the user name cannot be found, the **cuserid** subroutine returns a **NULL** pointer; if the *s* parameter is not a **NULL** pointer, then a null character ('\0') is stored into *s*[0].

Related Information

In this book: “getlogin” on page 3-196, “getpwent, getpwuid, getpwnam, setpwent, endpwent” on page 3-203, and “stdio” on page 3-297.

dbm

Purpose

Performs data base operations.

Library

Database Library (libdbm.a)

Syntax

```
int dbminit (file)  
char *file;
```

```
datum fetch (key)  
datum key;
```

```
int store (key, content)  
datum key, content;
```

```
int delete (key)  
datum key;
```

```
datum firstkey ( )
```

```
datum nextkey (key)  
datum key;
```

```
typedef struct  
{  
    char *dptr;  
    int  dsize;  
} datum;
```

Description

The **dbm** subroutines maintain a data base of *key-content* pairs. These subroutines can handle very large data bases and access keyed items in one or two file-system accesses.

The *key* parameter is a pointer to data specified by the *content* parameter. The sum of the sizes of the *key-content* pairs must not exceed the internal block size of 512 bytes. All *key-content* pairs that hash together must fit on a single block. The **store** subroutine returns an error if a disk block fills with inseparable data.

The *key* and the *content* parameters are described by the **typedef datum** structure. The **datum** structure sets up a string of bytes. The length of the string is specified by the **dsize** field. The string is pointed to by the **dptr** field. The **dptr** pointers that are returned by these subroutines point to static storage that changes with subsequent calls. You can use binary data or normal ASCII strings.

The data base is stored in two files. One file is a directory that contains a bit map and is suffixed with **.dir**. The second file contains all data and is suffixed with **.pag**. The **.pag** file contains holes that increases its apparent size to about four times its actual size. You

cannot copy a **.pag** file using the standard utilities such as **cp** and **cat** without first filling these holes.

Before you can access a data base, you must open the data base with the **dbminit** subroutine. The *file*, **.dir**, and **.pag** files must already exist before you call the **dbminit** subroutine. You can create an empty data base by creating zero-length **.dir** and **.pag** files.

After the data base is opened with the **dbminit** subroutine, you can use the **fetch** subroutine to access the data that is pointed to by the *key* parameter. You can use the **store** subroutine to write the data specified by the *content* parameter to a file and to specify the key to be used to access that data with the *key* parameter.

The **delete** subroutine removes the key specified by the *key* parameter and the data to which that key points. The **delete** subroutine does not actually reclaim the file space, but it does make it available for reuse.

The **firstkey** and **nextkey** subroutines make a linear pass through all of the keys in a data base. The **firstkey** subroutine returns the first key in the data base. The **nextkey** subroutine returns the next key in the data base. The following code makes a linear pass through a data base:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
{
    . . .
}
```

The order of keys that are presented to **firstkey** and **nextkey** depend on the hashing function.

Return Value

All of the **dbm** subroutines that return an *int* value return 0 upon successful completion, and they return a negative value if an error occurs. Subroutines that return a **datum** value indicate an error by setting the **dptr** field to **NULL**.

dosassign

Purpose

Assigns one DOS Services drive to another.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
int dosassign (drive, todrive)  
char *drive, *todrive;
```

Description

The **dosassign** subroutine causes all references to the drive specified by the *drive* parameter to use the drive specified by the *todrive* parameter. The *drive* and *todrive* parameters are strings containing the names of drives as configured by **dosinit** or in a user profile. The names can be in either uppercase or lowercase and must not include the colon.

If the *drive* parameter is **NULL**, all assignments are reset to their initial state.

Once assigned, using the drive specified by the *drive* parameter is equivalent to using the drive specified by the *todrive* parameter. However, the **dospwd** subroutine does not perform this translation. It returns a path name that includes either the drive name passed to it as a parameter or the drive name passed to the **doschdir** subroutine.

The **dosassign** subroutine does not change the current drive.

dosassign

Return Value

Upon successful completion, a value of 0 is returned. If the **dosassign** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “doschdir” on page 3-67 and “dospwd” on page 3-89.

Appendix b, “DOS Services Library” on page b-1.

doschdir

Purpose

Changes the current DOS Services directory or current drive.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
int doschdir (path)  
char *path;
```

Description

The **doschdir** subroutine changes the current directory on the current drive to the directory specified by the *path* parameter, or changes the current drive to the drive specified in the *path* parameter. If the *path* parameter contains only a drive name, then only the current drive is changed. If the *path* parameter contains only a directory path name, then only the current directory on the current drive is changed. If the *path* parameter contains both a drive name and a directory path name, then the current drive and the current directory are both changed.

When the current drive is set to a drive that contains a DOS file system, the AIX current directory cannot follow along. Therefore, the current AIX directory is set to the special directory **/usr/dos/nulldir**, if it exists.

Normally, the user does not have write access to the **/usr/dos/nulldir** directory. Therefore, if a program aborts, the core dumps are suppressed. If you do not want this to happen, you must remove the **/usr/dos/nulldir** directory.

doschdir

Return Value

Upon successful completion, a value of 0 is returned. If the **doschdir** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “dospwd” on page 3-89 and “dosassign” on page 3-65.
Appendix b, “DOS Services Library” on page b-1.

doschmod

Purpose

Changes the mode of a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
int doschmod (path, mode)
```

```
char *path;
```

```
long mode;
```

Description

The **doschmod** subroutine changes the mode of the file specified by the *path* parameter to the mode specified by the *mode* parameter. (For information about modes, see “doscreate” on page 3-71.)

Return Value

Upon successful completion, a value of 0 is returned. If the **doschmod** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “doscreate” on page 3-71 and “dosopen” on page 3-87.

Appendix b, “DOS Services Library” on page b-1.

dosclose

dosclose

Purpose

Closes a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>

int dosclose (dosfile)
DOSFILE dosfile;
```

Description

The **dosclose** subroutine closes the file descriptor specified by the *dosfile* parameter.

The *dosfile* parameter is a file descriptor obtained from a **dosopen**, **doscreate**, or **dosdup** subroutine.

Warning: DOS files are not implicitly closed when a process terminates. You must explicitly close all DOS files or you may lose data.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosclose** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “dosopen” on page 3-87, “doscreate” on page 3-71, and “dosdup” on page 3-73.

Appendix b, “DOS Services Library” on page b-1.

doscreate

Purpose

Creates a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
DOSFILE doscreate (path, mode)
```

```
char *path;
```

```
long mode;
```

Description

The **doscreate** subroutine creates a DOS file with the path and name specified by the *path* parameter. The newly created file has the flags set as specified by the *mode* parameter. If the file specified by the *path* parameter already exists, the file is truncated to zero length and the mode and owner are unchanged.

The *mode* parameter is a 32-bit word containing flags. The low-order 12 bits are access permission flags. (For information about access permission flags, see “chmod” on page 2-18.)

If the file is contained in a DOS file system, the write-by-owner bit of the *mode* parameter is the only significant access flag. If it is 0, the read-only flag is set in the DOS directory of the file created.

Mode flags for emulating functions unique to DOS are defined in the **dos.h** header file. They may be logically OR'ed together from the following list:

M-HIDDEN If this flag of the *mode* parameter is set, the file is created as a *hidden* file. If the file is created in a DOS file system, the appropriate bit is set in the directory. If the file is created in an AIX file system, the file name is prefixed by a . (period). If a hidden file is created with the same file name as an existing normal file, the normal file is renamed.

doscreate

M-SYSTEM If this flag of the *mode* parameter is set, and the file is created in a DOS file system, the **SYSTEM** attribute of the file is set. If the file is created in an AIX file system, this flag is ignored.

If the file is created in a DOS file system, the name of the file is translated to uppercase. If the file is created in an AIX file system, no translation takes place.

Warning: DOS files are not implicitly closed when a process terminates. You must explicitly close all DOS files or you may lose data.

Return Value

Upon successful completion, a non-NULL handle is returned. This handle is used in subsequent operations. The file is open for writing even if the mode does not permit writing. If the **doscreate** subroutine fails, a -1 is returned and **doserrno** is set to indicate the error.

The **doscreate** subroutine fails if one or more of the following are true:

- The user does not have write access to the directory containing the file.
- The user does not have write access to the file if the file already exists.
- The physical medium cannot be written to.
- No such device or address.
- No such device.

Related Information

In this book: “chmod” on page 2-18 and “dosopen” on page 3-87.

Appendix b, “DOS Services Library” on page b-1.

dosdup

Purpose

Duplicates a DOS Services file handle.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>

int dosdup (dosfile)
DOSFILE dosfile;
```

Description

The **dosdup** subroutine returns a new file descriptor that indicates the same file and has the same open flags as the original file descriptor. The *dosfile* parameter is a file descriptor returned by **dosread**, **doscreate**, or **dosdup**. The file position is initially set to the same value as the original, but changes independently. The file descriptor returned is the lowest one available.

Return Value

Upon successful completion, a DOS Services file handle is returned. If the **dosdup** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “doscreate” on page 3-71, “dosopen” on page 3-87, and “dosclose” on page 3-70.

Appendix b, “DOS Services Library” on page b-1.

dosexecve

dosexecve

Purpose

Executes a program with a DOS path name.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
int dosexecve (path, argv, envp)  
char *path, *argv[ ], *envp[ ];
```

Description

The **dosexecve** subroutine invokes an AIX-executable program identified by a DOS path name. The **dosexecve** subroutine corresponds to the **execve** system call except that the *path* parameter is interpreted with respect to the configured DOS file system.

The *path* parameter must identify an AIX-executable file. It cannot refer to a DOS-executable file, such as an **.EXE**, **.COM**, or **.BAT** file. See the instructions for running a **.BAT** file on page b-1.

The new program started by the **dosexecve** subroutine inherits the AIX run-time environment, which includes the AIX open file descriptors and other information. (See “fork” on page 2-44 and “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32 for a complete description of the AIX environment that is inherited after these system calls.)

However, the new program does *not* automatically inherit the DOS Services environment. See “dosunopen, dosreopen” on page 3-105 for details about passing the DOS Services environment to a child process.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosexecve** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “fork” on page 2-44, “wait” on page 2-176, “dosunopen, dosreopen” on page 3-105, and “dosinit” on page 3-80.

Appendix b, “DOS Services Library” on page b-1.

dosfirst, dosnext

dosfirst, dosnext

Purpose

Finds DOS files that match a pattern.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
char *dosfirst (srch, pattern, mode)
DOSFIND *srch;
char *pattern;
long mode;
```

```
char *dosnext (srch)
DOSFIND *srch;
```

Description

The **dosfirst** and **dosnext** subroutines return a pointer to a memory area containing a file name that matches the pattern specified by the *pattern* parameter and that has the attributes specified by the *mode* parameter.

The *pattern* parameter is a file path name that can contain the pattern-matching characters ? (question mark) and * (asterisk).

The *srch* parameter points to a **DOSFIND** structure. That same **DOSFIND** structure should be passed to the **dosnext** subroutine on subsequent uses of the **dosnext** subroutine.

The *mode* parameter contains flags that specify files to include in the search. If the *mode* parameter is 0, directories, hidden files, and DOS system files are omitted from the search. You can use the following flags OR'ed together in any combination:

- | | |
|-----------------|---|
| S_DIR | Includes directories in the search. |
| S_HIDDEN | Includes DOS/AIX hidden files in the search. |
| S_SYSTEM | Includes DOS system files in the search. |
| S_REG | Includes regular DOS files (all files other than directories, hidden files, or system files). |
| S_ALL | Includes all files (directories, hidden files, system files, and regular files). |

DOSFIND is defined in the **dos.h** header file and has the following format:

```
typedef long   DOSMODE;
typedef short  DOSFILE;

typedef struct
{
    long    seek;
    int     count;
    long    *disk;
    int     mode;
    int     tnxtcl;
} dosrch;

typedef struct
{
    long    mode;
    char    path[128];
    char    *base;
    char    *extn;
    char    is_dos;
    DOSFILE handle;
    short   index;
    dosrch  dos_srch;
} DOSFIND;
```

A return of **NULL** from either subroutine indicates that no more files matching the pattern can be found. If the search is terminated before the **NULL** return occurs, you should use the **free** subroutine to free the memory area returned from the last call.

Related Information

Appendix b, "DOS Services Library" on page b-1.

dosfsync

dosfsync

Purpose

Synchronizes a specified DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>

int dosfsync (dosfile)
DOSFILE dosfile;
```

Description

The **dosfsync** subroutine guarantees that any changes to the file specified by the *dosfile* parameter has been written to the device on which the file exists when the subroutine returns. The use of the **dosfsync** subroutine has no detectable effect in a single process that runs to completion. It is useful in multi-processing applications and as a form of backup.

The *dosfile* parameter is an open file descriptor that was obtained from a **dosopen**, **doscreate**, or **dosdup** subroutine.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosfsync** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “fsync” on page 2-46, “dosopen” on page 3-87, “doscreate” on page 3-71, “dosdup” on page 3-73, and “doswrite” on page 3-109.

Appendix b, “DOS Services Library” on page b-1.

dosinit

dosinit

Purpose

Initializes the DOS Services environment.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
int dosinit ( )
```

Description

The **dosinit** subroutine initializes the run-time environment required by the DOS Services subroutines. Programs must call the **dosinit** subroutine before using any of the DOS Services subroutines.

The **dosinit** subroutine looks for the initialization information in the following places:

1. It attempts to read the information from the file named by **DOSENV**. **DOSENV** is a variable in the AIX environment. The **dos** command (or any parent process) creates this file by calling the **dosunopen** subroutine before starting a child process.
2. If the **DOSENV** variable is not set, then **dosinit** configures DOS Services based on the AIX environment variables in the following list.
3. For the AIX variables that are not defined, the default values shown in the following are used.

AIX Variable	Default Value
DOS_NUL	/dev/null
DOS_CON	/dev/tty
DOS_COM1	/dev/tty0
DOS_COM2	/dev/tty1
DOS_AUX	/dev/tty0
DOS_LP0	/dev/lp0

DOS_LP1	/dev/lp1
DOS_LP2	/dev/lp2
.	.
.	.
.	.
DOS_LP7	/dev/lp7
DOS_A	/dev/fd0
DOS_B	/dev/fd0
DOS_C	\$HOME
DOS_D	/
DOS_E	No default
DOS_F	No default
.	.
.	.
.	.
DOS_Z	No default.

Note: If **\$HOME** is set to /, or if it is undefined, then the default directory for **DOS_C** is **/usr/dos**.

The value assigned to each of these variables must be either:

- The full AIX path name of an accessible device, file, or directory, or
- A string in the form **|command**, where *command* is an AIX command. The specified command receives, as its standard input, the data that DOS Services programs write to the corresponding logical device.

The **dosinit** subroutine also uses the following environment variable:

DOSDISK	Specifies the initial default drive for DOS Services. If DOSDISK is not set, then dosinit searches sequentially from A: to Z: and sets the default drive to the first valid file system found.
----------------	--

The AIX environment variables that specify the DOS Services configuration can be set in the system **/etc/profile** file, in the user's **\$HOME/.profile**, or directly from the AIX command line. See the **sh** command in *AIX Operating System Commands Reference* for more information about setting AIX environment variables, which are also called **shell variables**.

The AIX environment variables are accessed only once during a single session started by the **dos** command. From then on, internal configuration tables are used.

Return Value

Upon successful completion **dosinit** returns a value of 0. Otherwise a value of -1 is returned, and **doserrno** is set to indicate the error.

Related Information

In this book: “dosunopen, dosreopen” on page 3-105, “dosread” on page 3-91, and “doswrite” on page 3-109

Appendix b, “DOS Services Library” on page b-1.

The **dos** and **sh** commands in *AIX Operating System Commands Reference*.

dosmkdir

Purpose

Creates a directory.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>

int dosmkdir (path)
char *path;
```

Description

The **dosmkdir** subroutine creates a directory using the path specified by the *path* parameter. All components of the *path* parameter except the last component must already exist.

If creating an AIX directory, the **dosmkdir** subroutine forks and executes the AIX **/bin/mkdir** command, which creates the directory. This is done because only a process with an effective user ID of superuser can create an AIX directory.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosmkdir** subroutine fails, then it returns a nonzero value and sets **doserrno** to indicate the error.

Certain failures of the **mkdir** command also cause an error message to be written to the standard error output.

Related Information

In this book: “dosrmdir” on page 3-95.

Appendix b, “DOS Services Library” on page b-1.

The **mkdir** command in *AIX Operating System Commands Reference*.

dosmktemp

Purpose

Creates a DOS temporary file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
DOSFILE dosmktemp ( )
```

Description

The **dosmktemp** subroutine creates and opens a temporary file. The file is open for reading and writing. The access permission flags are set so that the owner has read-write permission and all others have no permission.

The file is created in the **/tmp** directory with a name that includes the process ID of the running process and a serial number. The **/tmp** directory does not need to be accessible through the DOS file system configuration.

Return Value

Upon successful completion, a non-NULL file handle is returned. This handle is to be used in subsequent operations.

If the **dosmktemp** subroutine fails, -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “dosopen” on page 3-87 and “doscreate” on page 3-71.

Appendix b, “DOS Services Library” on page b-1.

dosopen

Purpose

Opens a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
DOSFILE dosopen (path, oflag, mode)  
char *path;  
int oflag;  
long mode;
```

Description

The **dosopen** subroutine opens the file specified by the *path* parameter. The *oflag* parameter specifies the type of open. The *mode* parameter specifies the access mode of the file if a new file is created.

The *oflag* parameter is constructed by logically OR-ing one or more of the following values:

- | | |
|------------------|--|
| DO_RDONLY | Open for reading only. |
| DO_WRONLY | Open for writing only. |
| DO_RDWR | Open for reading and writing. |
| DO_APPEND | If set, the file pointer is set to the end of the file prior to each write operation. |
| DO_CREAT | If the file does not exist, create it. Use <i>mode</i> to establish the protection mode of the new file. (For information on creating a DOS file, see "doscreate" on page 3-71.) |
| DO_TRUNC | If the file exists, truncate it. Otherwise begin writing at the end of file. |

dosopen

DO_EXCL If the file already exists, the **dosopen** subroutine fails.

DO_ASCII Interpret the file as an ASCII text file. (For information about ASCII files, see “dosread” on page 3-91 and “doswrite” on page 3-109.)

Note: Only one of **DO_RDONLY**, **DO_WRONLY**, and **DO_RDWR** can be specified. The others can be used in any combination.

If the file being opened is on a DOS file system, the name file name given is translated to uppercase. If the file system is an AIX file system, no translation takes place.

Warning: DOS files are not implicitly closed when a process terminates. You must explicitly close all DOS files or you may lose data.

Return Value

Upon successful completion, a DOS Services file handle is returned. This handle is used in subsequent operations. If the **dosopen** subroutine fails, -1 value is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “doscreate” on page 3-71, “dosclose” on page 3-70, “dosdup” on page 3-73, “dosread” on page 3-91, “dosseek” on page 3-97, and “doswrite” on page 3-109.

Appendix b, “DOS Services Library” on page b-1.

dospwd

Purpose

Gives the full path name of the current directory.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>

char *dospwd (drive)
char *drive;
```

Description

The **dospwd** subroutine returns a pointer to the area of memory that contains the null-terminated name of the current directory for the drive specified by the *drive* parameter. If the *drive* parameter is a valid drive name (such as "A:" or "B:"), then the current directory for that drive is returned. If the *drive* parameter is **NULL**, then the current directory for the current drive is returned.

Return Value

Upon successful completion, a pointer to a string specifying the full path name of the current directory is returned. The memory for the name is allocated with the **malloc** subroutine and should be deallocated with the **free** subroutine. If the **dospwd** subroutine fails, a **NULL** pointer is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “doschdir” on page 3-67 and “dosassign” on page 3-65.

Appendix b, “DOS Services Library” on page b-1.

dosread

Purpose

Reads from a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
int dosread (dosfile, buf, n)  
DOSFILE dosfile;  
char *buf;  
int n;
```

Description

The **dosread** subroutine reads the number of bytes specified by the *n* parameter. The *dosfile* parameter specifies the handle of the file from which the bytes are to be read. The bytes read from the file are written into the buffer pointed to by the *buf* parameter.

Reading begins from the current position in the file. The current position of the file is incremented by the number of bytes read.

If the **DO_ASCII** flag was set when the file was opened, and if the file is located on a DOS file system, then the **dosread** subroutine translates the DOS ASCII data to AIX ASCII format. If **DO_ASCII** was not set when the file was opened, or if the file is located on an AIX file system, then this translation does not take place.

When DOS ASCII translation is being performed, the **dosread** subroutine removes the ASCII CR characters, thus changing the CR-LF sequence used by DOS to the '\n' (new-line character) that AIX uses. An end-of-file condition occurs when **Ctrl-Z** is encountered. The **dosread** subroutine does not return the **Ctrl-Z** character as part of the data.

dosread

Return Value

Upon successful completion, the number of bytes actually read (after DOS translation, if any) returned. If an end-of-file is read, a value of 0 is returned. If the **dosread** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “dosopen” on page 3-87, “doscreate” on page 3-71, “doswrite” on page 3-109, “dosclose” on page 3-70, and “dosseek” on page 3-97.

Appendix b, “DOS Services Library” on page b-1.

dosrename

Purpose

Renames a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>

int dosrename (oldpath, newfile)
char *oldpath, *newfile;
```

Description

The **dosrename** subroutine changes the name of a DOS file. The file to be renamed is specified by the path name pointed to by the *oldpath* parameter. The new name for that file is specified by the *newfile* parameter. The *newfile* parameter must be a simple file name and optional extension.

The **dosrename** subroutine must not be used to move a file from one directory to another directory. The **dosrename** subroutine fails if the file name specified by the *newfile* parameter already exists in the directory specified by the *oldpath* parameter.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosrename** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “doscreate” on page 3-71, “dosopen” on page 3-87, and “dosunlink” on page 3-103.

Appendix b, “DOS Services Library” on page b-1.

dosrmdir

Purpose

Removes a DOS Services directory.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>

int dosrmdir (path)
char *path;
```

Description

The **dosrmdir** subroutine removes the directory specified by the *path* parameter. A directory must be empty before it can be removed. To remove a directory, the current process must have write access permission to the directory and to its parent directory.

If an AIX directory is to be removed, the **dosrmdir** subroutine forks and executes the AIX **/bin/rmdir** command, which removes the directory. This is done because only a process with an effective user ID of superuser can remove an AIX directory.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosrmdir** subroutine fails, it returns a nonzero value and sets **doserrno** to indicate the error. Certain failures of the **rmdir** command also cause an error message to be written to the standard error output.

Related Information

In this book: “dosmkdir” on page 3-83.

Appendix b, “DOS Services Library” on page b-1.

The **rmdir** command in *AIX Operating System Commands Reference*.

dosseek

Purpose

Moves the DOS file read/write pointer.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
int dosseek (dosfile, offset, whence)  
DOSFILE dosfile;  
long offset;  
int whence;
```

Description

The **dosseek** subroutine moves the current position pointer in the file specified by the *dosfile* parameter.

- If the *whence* parameter is 0, the pointer is set to the position specified by the *offset* parameter.
- If the *whence* parameter is 1, the pointer is incremented by the number of bytes specified by the *offset* parameter.
- If the *whence* parameter is 2, the pointer is set to the size of the file plus the number of bytes specified by the *offset* parameter.

If the **dosseek** subroutine has been issued on a file that was opened with the **DO_ASCII** flag set, the *offset* parameter must be 0. Otherwise, the results are unpredictable.

Return Value

Upon successful completion, the file pointer value is returned. If the **dosseek** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Diagnostics

The **dosseek** subroutine fails and the file pointer remains unchanged if one or more of the following are true:

- EBADF** *dosfile* is not an open file descriptor.
- ESPIPE** *dosfile* is associated with a pipe or FIFO.
- EINVAL** *whence* is not 0, 1 or 2. This also causes a **SIGSYS** signal.
- EINVAL** The resulting file pointer would be negative.

Related Information

In this book: “dosopen” on page 3-87, “doscreate” on page 3-71, and “dosdup” on page 3-73.

Appendix b, “DOS Services Library” on page b-1.

dosstat, dosfstat

Purpose

Gets the status of a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
int dosstat (path, buf)  
char *path;  
DOSSTAT *buf;
```

```
int dosfstat (dosfile, buf)  
DOSFILE dosfile;  
DOSSTAT *buf;
```

Description

The **dosstat** subroutine gets the status of the file named by the *path* parameter and stores that status into the memory area pointed to by the *buf* parameter. Read, write, and execute permissions are not required of the named file, but all directories included in the path name must be searchable.

The **dosfstat** subroutine gets the status of an open file named by the *dosfile* parameter and stores that status into the memory area pointed to by the *buf* parameter.

The **DOSSTAT** structure is defined in the **dos.h** file and has the following form:

```
typedef struct  
{  
    char    st_drive_id, st_filetype;  
    long    st_mode, st_ino, st_dev, st_rdev,  
           st_nlink, st_uid, st_gid, st_size;  
    time_t  st_atime, st_mtime, st_ctime;  
} DOSSTAT;
```

dosstat, dosfstat

The **st_drive_id** field is set to the logical drive on which the file exists. The **st_filetype** field is set to the following values:

- u For an AIX file.
- d For a DOS file.
- t For a tty file.
- o For any other type of file.

For non-DOS files, all other fields are identical to the fields defined in “stat.h” on page 5-61. For DOS files, the fields have the following meanings:

- st_mode** If the **M_DIRECTORY** bit is set, the file is a directory. All other bits correspond to the *mode* bits as defined in the **doscreate** subroutine.
- st_ino** The number of the first cluster in the file.
- st_rdev** Contains the attribute byte from the DOS directory entry.
- st_size** The size in bytes of the file as set in the DOS directory entry.
- st_mtime** The time the file was last modified, in AIX format. (For information about this format, see “time” on page 2-158 and “ctime, localtime, gmtime, asctime, tzset” on page 3-43.)
- st_dev** Meaningless for DOS files.
- st_nlink** Meaningless for DOS files.
- st_uid** Meaningless for DOS files.
- st_gid** Meaningless for DOS files.
- st_atime** Meaningless for DOS files.
- st_ctime** Meaningless for DOS files.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosstat** or **dosfstat** subroutines fail, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “dosustat” on page 3-107 and “stat, fstat” on page 2-153.

Appendix b, “DOS Services Library” on page b-1.

dostouch

Purpose

Changes the modification date of a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
int dostouch (path, date)  
char *path;  
long date;
```

Description

The **dostouch** subroutine changes the time of the last modification of the file specified by the *path* parameter to the time specified by the *date* parameter. The *time* parameter contains a time in a format like that returned by the **time** system call (see “time” on page 2-158).

If the *path* parameter identifies a file on an AIX file system, the program **/bin/touch** is invoked to change the time.

Return Value

Upon successful completion, a value of 0 is returned. If the **dostouch** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “time” on page 2-158, “dosstat, dosfstat” on page 3-99, and “ctime, localtime, gmtime, asctime, tzset” on page 3-43.

Appendix b, “DOS Services Library” on page b-1.

dosunlink

Purpose

Unlinks (deletes) a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>

int dosunlink (path)
char *path;
```

Description

The **dosunlink** subroutine deletes the directory entry named by the *path* parameter.

The file is removed from the file system if it is a DOS file or if it is an AIX file with no other links to it.

The **dosunlink** subroutine fails if the invoking process does not have write access to the file being removed and to the directory in which it is contained. It is successful if the file is in use, but the file is not deleted until all users have closed it.

The directory entry is removed upon return to **dosunlink**, but all users who had the file open when **dosunlink** was called still have a local copy for their use.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosunlink** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “dosopen” on page 3-87, “doscreate” on page 3-71, and “dosrename” on page 3-93.

Appendix b, “DOS Services Library” on page b-1.

dosunopen, dosreopen

Purpose

Passes the DOS Services environment from a parent process to a child process.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
int dosunopen ( )
```

```
int dosreopen ( )
```

Description

The **dosunopen** and **dosreopen** subroutines provide a means to pass the current DOS Services environment to a child process. The following sequence accomplishes this task:

1. The parent process calls the **dosunopen** subroutine to construct a file that describes the current state of the DOS Services environment. **dosunopen** also sets the **DOSENV** variable (in the AIX environment) to the name of this file.
2. The parent process then issues a **fork** system call to start a child process. The child process automatically inherits copies of its parent's AIX open file descriptors, but not the DOS file descriptors.
3. The child process calls the **dosexecve** subroutine (or one of the **exec** system calls) to run a new program. This program calls the **dosinit** subroutine, which initializes the child's DOS Services environment from the file named by the **DOSENV** variable. The child process now has access to the DOS files opened by its parent.
4. The parent process invokes the **wait** system call to wait for the child to finish running its program.
5. When the child has finished, the parent process calls the **dosreopen** subroutine to reopen the original DOS Services environment and to delete the environment file created by **dosunopen**.

dosunopen, dosreopen

If a program attempts to reopen a file on a removable diskette and that diskette is not inserted into the machine, then the **dosreopen** subroutine prompts you to reinsert the correct diskette.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosunopen** or **dosreopen** subroutine fails, then it returns a value of -1 and sets **doserrno** to indicate the error.

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “fork” on page 2-44, “wait” on page 2-176, “dosinit” on page 3-80, “dosopen” on page 3-87, “dosexecve” on page 3-74, and “getenv” on page 3-193.

Appendix b, “DOS Services Library” on page b-1.

dosustat

Purpose

Gets the status of a given DOS Services device.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>
```

```
int dosustat (device, buf)
char *device;
DOSUSTAT *buf;
```

Description

The **dosustat** subroutine gets the status of the DOS Services device specified by the *device* parameter and returns that information to the area of memory specified by the *buf* parameter. The *device* parameter is a pointer to a string that contains the name of a DOS Services device.

The **DOSUSTAT** structure is defined in **/usr/include/dos.h** and has the following format:

```
typedef struct
{
    char  upath[128];    /* Device or file specified in configuration */
    char  umount[128];   /* AIX directory mounted on it, or NULL */
    char  volume[32];    /* Volume name */
    int   freespace;     /* Number of free bytes */
    char  fstype;        /* AIX file system='u', DOS file system='d' */
} DOSUSTAT;
```

The **umount** structure member is not **NULL** only when the *upath* field identifies a device name (such as **/dev/fd0**), and that device contains a mountable AIX file system.

dosustat

The **volume**, **freespace**, and **fstype** members are set only when the *device* parameter is a disk name (a single letter followed by a colon). If the *device* is a removable disk, **umount** may read it to determine if it is a valid DOS or AIX file system.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosustat** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “ustat” on page 2-172 and “dosstat, dosfstat” on page 3-99.

Appendix b, “DOS Services Library” on page b-1.

doswrite

Purpose

Writes to a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>

int doswrite (dosfile, buf, n)
DOSFILE dosfile;
char *buf;
int n;
```

Description

The **doswrite** subroutine writes the number of bytes specified by the *n* parameter to the file whose handle is specified by the *dosfile* parameter. The bytes are written from the buffer pointed to by the *buf* parameter.

Writing begins from the current position in the file. The current position is incremented by the number of bytes written.

If the file was opened with the **DO_APPEND** flag set, then the file pointer is set to the end of the file prior to each write.

If the **DO_ASCII** flag was set when the file was opened, and if the file is located on a DOS file system, then the **doswrite** subroutine translates the AIX ASCII data to DOS ASCII format. If **DO_ASCII** was not set when the file was opened, or if the file is located on an AIX file system, then this translation does not take place.

When DOS ASCII translation is being performed, the **doswrite** subroutine inserts an ASCII CR character before each LF character, thus changing each AIX '\n' (new-line character) to the CR-LF sequence that DOS uses to indicate a new line. A Ctrl-Z character is written as the last character when the file is closed.

doswrite

If a write requests that more translated bytes be written than there is room for in the file, then only as many bytes as there is room for are written. The next request to write more than zero bytes returns an error.

Return Value

Upon successful completion, a number equal to the number of bytes written before translation is returned. If the **doswrite** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “dosopen” on page 3-87, “doscreate” on page 3-71, “dosread” on page 3-91, “dosclose” on page 3-70, and “dosseek” on page 3-97.

Appendix b, “DOS Services Library” on page b-1.

drand48

Purpose

Generates uniformly distributed pseudo-random number sequences.

Library

Standard C Library (**libc.a**)

Syntax

double drand48 ()

double erand48 (*xsubi*)
unsigned short *xsubi*[3];

long lrand48 ()

long nrand48 (*xsubi*)
unsigned short *xsubi*[3];

long mrand48 ();

long jrand48 (*xsubi*)
unsigned short *xsubi*[3];

void srand48 (*seedval*)
long *seedval*;

unsigned short *seed48 (*seed16v*)
unsigned short *seed16v*[3];

void lcong48 (*param*)
unsigned short *param*[7];

Description

This family of subroutines generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic.

The **drand48** and **erand48** subroutines return nonnegative double-precision floating-point values uniformly distributed over the range of y values such that $0.0 \leq y < 1.0$.

The **lrand48** and **nrand48** subroutines return nonnegative long integers uniformly distributed over the range of y values such that $0 \leq y < 2^{31}$.

The **mrnd48** and **jrand48** subroutines return signed long integers uniformly distributed over the range of y values such that $-2^{31} \leq y < 2^{31}$.

The **srand48**, **seed48** and **lcong48** subroutines initialize the random-number generator. Programs should invoke one of them before calling **drand48**, **lrand48** or **mrnd48**.

(Although it is not recommended practice, constant default initializer values are supplied automatically if the **drand48**, **lrand48** or **mrnd48** subroutines are called without first

calling an initialization subroutine.) The **erand48**, **nrand48** and **jrand48** subroutines do not require that an initialization subroutine to be called first.

All the subroutines work by generating a sequence of 48-bit integer values, X_i , according to the linear congruential formula:

$$X_{n+1} = (aX_n + c)_{\text{mod } m} \quad n \geq 0$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless the **lcong48** subroutine has been called, the multiplier value a and the addend value c are:

$$a = 5DEECE66D_{16} = 273673163155_8$$

$$c = B_{16} = 13_8$$

The value returned by the **drand48**, **erand48**, **lrand48**, **nrand48**, **mrnd48**, and **jrand48** subroutines is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (most significant) bits of X_i and transformed into the returned value.

The **drand48**, **lrand48** and **mrnd48** subroutines store the last 48-bit X_i generated into an internal buffer; that is why they must be initialized prior to being invoked.

The **erand48**, **nrand48** and **jrand48** subroutines require the calling program to provide storage for the successive X_i values in the array pointed to by the *xsubi* parameter. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as a parameter.

By using different parameters, the **erand48**, **nrand48**, and **jrand48** subroutines allow separate modules of a large program to generate several *independent* sequences of pseudo-random numbers. In other words, the sequence of numbers that one module generates does *not* depend upon how many times the subroutines are called by other modules.

The initializer subroutine **srand48** sets the high-order 32 bits of X_i to the 32 bits contained in its parameter. The low order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer subroutine **seed48** sets the value of X_i to the 48-bit value specified in the array pointed to by the *seed16v* parameter. In addition, **seed48** returns a pointer to a 48-bit internal buffer that contains the previous value of X_i . that is used only by **seed48**. The returned pointer allows you to restart the pseudo-random sequence at a given point. Use the pointer to copy the previous X_i value into a temporary array. Later you can call **seed48** with a pointer to this array to resume where the original sequence left off.

The **lcong48** subroutine specifies the initial X_i value, the multiplier value a , and the addend value c . The parameter array elements *param*[0-2] specify X_i , *param*[3-5] specify the multiplier a , and *param*[6] specifies the 16-bit addend c . After **lcong48** has been called, a subsequent call to either **srand48** or **seed48** restores the standard a and c as specified previously.

Related Information

In this book: “rand, srand” on page 3-273.

ecvt, fcvt, gcvt

ecvt, fcvt, gcvt

Purpose

Converts a floating-point number to a string.

Library

Standard C Library (**libc.a**)

Syntax

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
```

```
char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
```

```
char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

Description

The **ecvt**, **fcvt**, and **gcvt** subroutines convert floating-point numbers to strings.

The **ecvt** subroutine converts the *value* parameter to a null-terminated string and returns a pointer to it. The *ndigit* parameter specifies the number of digits in the string. The low-order digit is rounded. **ecvt** sets the **int** pointed to by the *decpt* parameter to the position of the decimal point relative to the beginning of the string. (A negative number means the decimal point is to the left of the digits given in the string). The decimal point itself is not included in the string. The **ecvt** subroutine also sets the **int** pointed to by the *sign* parameter to a non-zero value if the *value* parameter is negative, and sets it to 0 otherwise.

The **fcvt** subroutine functions identically to **ecvt**, except that it rounds the correct digit for outputting *ndigit* digits in FORTRAN F-format.

The **gcvt** subroutine converts the *value* parameter to a null-terminated string, stores it in the array pointed to by the *buf* parameter, and then returns *buf*. **gcvt** attempts to produce a string of *ndigit* significant digits in FORTRAN F-format. If this is not possible, then E-format is used. **gcvt** suppresses trailing zeroes. The string is ready for printing, complete with minus sign, decimal point, or exponent, as appropriate.

The **ecvt** and **fcvt** subroutines represent the special floating-point values that are specified in the ANSI/IEEE Std 754-1985 standard as follows:

Quiet NaN	QNaN
Signalling NaN	SNaN
$\pm \infty$	INF

The sign associated with each of these values is stored into the *sign* parameter. Note, also, that 0 can be positive or negative.

Warning: All three subroutines store the strings in a static area of memory whose contents are overwritten each time one of the subroutines is called.

Related Information

In this book: “a64l, l64a” on page 3-4, “frexp, ldexp, modf” on page 3-179, “printf, fprintf, sprintf” on page 3-262, and “scanf, fscanf, sscanf” on page 3-281.

end, etext, edata

end, etext, edata

Purpose

Defines the last location of a program.

Library

None

Syntax

```
extern end;  
extern etext;  
extern edata;
```

Description

The external names **end**, **etext**, and **edata** are defined by the loader for all programs. They are not subroutines, but identifiers associated with the following addresses:

etext The first address following the program text
edata The first address following the initialized data region
end The first address following the data region that is not initialized.

The *program break* is the first location beyond the data. When a program begins running, this location coincides with **end**. However, many factors can change the program break, including:

- The **brk** system call
- The **malloc** subroutine
- The standard input/output subroutines
- The **-p** flag on the **cc** command.

Therefore, use **sbrk(0)**, not **end**, to determine the program break.

Related Information

In this book: “brk, sbrk” on page 2-14, “malloc, free, realloc, calloc” on page 3-224, and “stdio” on page 3-297.

The **cc** command in *AIX Operating System Commands Reference*.

Purpose

Computes the error and complementary error functions.

Library

Math Library (**libm.a**)

Syntax

```
#include <math.h>
```

```
double erf (x)  
double x;
```

```
double erfc (x)  
double x;
```

Description

The **erf** subroutine returns the error function of x , defined as:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

The **erfc** subroutine returns $1.0 - \text{erf}(x)$. The **erfc** subroutine is provided because of the extreme loss of relative accuracy if **erf**(x) is called for large values of x and the result is subtracted from 1.0. For example, 12 decimal places are lost when calculating $(1.0 - \text{erf}(5))$.

Related Information

In this book: “exp, log, log10, pow, sqrt” on page 3-121.

errunix

Purpose

Logs application errors.

Library

Run-time Services Library (**librts.a**)

Syntax

```
int errunix (buf, cnt)
char *buf;
unsigned int cnt;
```

Description

The **errunix** subroutine invokes the application error device driver to record an error log entry. **errunix** is a C run-time subroutine. Device drivers should use the **errsav** subroutine to log error messages.

If the error device driver is not open, **errunix** opens it. Then the error log entry is written to it.

The *buf* parameter points to a buffer that contains the following information:

1. A word (**int**) that contains the **class**, **subclass**, **mask**, and **type** of the message, as defined in the discussion of "error" on page 6-11
2. An **int** that specifies the number of words of dependent data for the error log entry, including this **int** itself
3. Words that contain the dependent information for the error log entry. The number of dependent data words must be one less than the word count specified immediately before them.

The other fields of the error log header (length, date and time, time extended, node name, and virtual machine ID) are supplied for you automatically.

The *cnt* parameter specifies the number of bytes in the buffer pointed to by *buf*. *cnt* must be a multiple of 4.

Return Value

Upon successful completion, a value of 0 is returned. If the **errunix** subroutine fails, an error message is written to the standard error output, and a value of -1 is returned.

File

`/dev/error`

Related Information

In this book: “errsave” on page e-6, and “error” on page 6-11

exp, log, log10, pow, sqrt

Purpose

Computes exponential, logarithm, power, and square root functions.

Library

Math Library (**libm.a**)

Syntax

```
#include <math.h>
```

```
double exp (x)  
double x;
```

```
double log (x)  
double x;
```

```
double log10 (x)  
double x;
```

```
double pow (x, y)  
double x, y;
```

```
double sqrt (x)  
double x;
```

Description

The **exp** subroutine returns e^x . If the correct value overflows, **exp** returns **HUGE** and sets **errno** to **ERANGE**.

The **log** subroutine returns the natural logarithm of x , $\ln x$. The value of x must be positive. If x is negative or 0, then **log** returns the value **-HUGE**, sets **errno** to **EDOM**, and writes an error message to the standard error output.

The **log10** subroutine returns the logarithm base 10 of x , $\log_{10} x$. The value of x must be positive. If x is negative or 0, then **log10** returns the value **-HUGE**, sets **errno** to **EDOM**, and writes an error message to the standard error output.

The **pow** subroutine returns x^y . The values of x and y may not both be 0. If x is nonpositive, then y must be an integer. If x is nonpositive and y is not an integer, or if x and y are both 0, then **pow** returns the value 0, sets **errno** to **EDOM**, and writes an error message to the standard error output. If the correct value overflows, **pow** returns **HUGE** and sets **errno** to **ERANGE**.

exp, log, log10, pow, sqrt

The **sqrt** subroutine returns the square root of x . The value of x can not be negative. If x is negative, then **sqrt** returns the value 0 sets **errno** to **EDOM**, and writes an error message to the standard error output.

These error-handling procedures can be changed with the **matherr** subroutine.

Related Information

In this book: “hypot” on page 3-213, “matherr” on page 3-226, and “sinh, cosh, tanh” on page 3-292.

Extended Curses

Purpose

Controls cursor movement and windowing.

Library

Extended Curses Library (**libcur.a**)

Syntax

```
#include <cur01.h>
```

Description

The Extended Curses subroutines control input and output to a work station, performing optimized cursor movement, windowing, and other functions. This package is based on the **curses** subroutine package, which is included in most UNIX-compatible systems. The **curses** subroutines are also included in AIX for complete compatibility with existing programs (see “curses” on page 3-48).

The enhancements provided by Extended Curses include:

- A wider range of display attributes
- Generalized drawing of boxes
- Terminal-independent input data processing
- Extended window control
- Pane, panel, and field concepts.

Terminology

- window** The internal representation of what a portion of the display may look like at some point in time. Windows can be any size from the entire display screen to a single character.
- screen** A window that is large as the display screen. A screen named **stdscr** is automatically provided.
- terminal** Sometimes called a **terminal screen**. A special screen that is the Extended Curses package's understanding of what the work station's display screen currently looks like. The terminal screen is identified by a window named

Extended Curses

curscr, which should not be accessed directly by the user. Instead, changes should be made to **stdscr** (or a user-defined screen) and then **refresh** (or **wrefresh**) should be called to update the terminal.

presentation space

The array that contains the data and attributes associated with a window.

pane

An area of the display that shows all or part of the data contained in a presentation space associated with that pane.

panel

A group of one or more panes that are treated as a unit. The panes of a panel are displayed together, erased together, and usually represent a unit of information to a person using the application. A panel is represented on the display as a rectangular area that is tiled (completely filled) with panes.

field

An area in a presentation space into which the program accepts input.

See the discussion of Extended Curses in *AIX Operating System Programming Tools and Interfaces* for more detailed information about these concepts.

Linking the Extended Curses Routines

The Extended Curses routines also call **terminfo** subroutines, which are located in the Curses Library (**libcurses.a**). Therefore, compile programs that use Extended Curses routines with the flags **-lcur -lcurses**.

Header Files

- The **cur00.h** header file replaces **curses.h** when converting programs that use the original **curses** package to Extended Curses.
- All of the routines require the **cur01.h** header file.
- The key codes returned by **getch** are defined in **cur02.h**.
- The **cur03.h** header file defines attribute priority codes, and is not needed by application programs.
- The **unctrl** routine requires **cur04.h**
- The routines that manage panes and panels (the routines whose names begin with **ec**) also require the **cur05.h** header file.

Naming Conventions

The new routines added to the original **curses** package begin with the letters **ec**.

Many routines operate on **stdscr**, the standard screen, by default. Corresponding routines that allow you to specify a window have the same name, prefixed with the letter **w**. For example, **addch** adds a character to **stdscr**, while **waddch** allows you to specify the window. Sometimes a routine beginning with **p** also exists, such as **paddch**, which allows you to specify a pane.

Some routines also allow you to specify cursor movement with the action to be performed. These routines have a prefix of **mv**. Thus, **addch** becomes **mvaddch**, **waddch** becomes **mvwaddch**, and **paddch** becomes **mvpaddch**. Each of these routines is equivalent to calling **move** or **wmove** before performing the operation.

The various prefixed forms of the routines are implemented as macros. In each case, the routine beginning with **w** is the base subroutine from which the others are defined.

Parameters

The following declarations serve for all of the routines:

```
char ch, *string;
int line, col, firstline, firstcol;
int numlines, numcols, numchars, mode;
bool boolf;
WINDOW *win, *win1, *win2, *oldwin, *newwin;
PANE *pane;
PANEL *panel;
```

Return Values

Unless otherwise noted, each routine returns a value of type **int** that is either **OK** (indicating successful completion) or **ERR** (if an error is encountered).

The Extended Curses Routines

The Extended Curses routines are listed here alphabetically, except that routines with **w**, **p**, and **mv** prefixes are listed with the corresponding routine that does not have these prefixes.

addch (*ch*)
waddch (*win*, *ch*)
paddch (*pane*, *ch*)
mvaddch (*line*, *col*, *ch*)
mvwaddch (*win*, *line*, *col*, *ch*)
mvpaddch (*pane*, *ch*)

The **addch** routine adds the character specified by the *ch* parameter on the window at the current (*line*, *col*) coordinates. **paddch** adds the character to the presentation space for the pane specified by the *pane* parameter. If the character is '\n' (new-line character), the line is cleared to the end, and the current (*line*, *col*) coordinates will be changed to the beginning of the next line. A '\r' (return character) moves the current position to the beginning of the current line on the window. A '\t' (tab character) is expanded into spaces in the normal tabstop positions of every eight columns.

Adding a character to the lower right corner of a window that includes the lower right corner of the display causes many terminals to scroll the entire display image up one line. If adding a character or a character attribute would cause such scrolling to occur, then **addch** makes the change on the window, but does not mark it for **wrefresh** purposes; in this case **addch** returns the value **ERR**.

addstr (*string*)
waddstr (*win*, *string*)
paddstr (*pane*, *string*)
mvaddstr (*line*, *col*, *string*)
mvwaddstr (*win*, *line*, *col*, *string*)
mvpaddstr (*pane*, *line*, *col*, *string*)

The **addstr** routine adds the string pointed to by the *string* parameter on the window at the current (*line*, *col*) coordinates.

Upon successful completion, **addstr** returns **OK** and the current (*line*, *col*) coordinates point to the location just beyond the end of the string. The **addstr** routine returns **ERR** if an attempt is made to add a character to the lower right corner of a window that includes the lower right corner of the display. In this case, **addstr** writes as much of the string on the window as possible.

beep ()

The **beep** routine sounds the work station's speaker or bell.

box (*win, vert, hor*)

char *vert, hor;*

The **box** routine draws a box around the window specified by the *win* parameter. The box uses the character specified by the *vert* parameter to draw the vertical sides of the box, and the character specified by the *hor* parameter for drawing the horizontal lines and corners.

If the window includes the lower right corner of the display and **scrollok** is not set, then the lower right corner of the box is not shown on the window and the **box** routine returns **ERR**.

The **box** routine is a macro that invokes **superbox**.

cbox (*win*)

The **cbox** routine draws a box around the window specified by the *win* parameter. The characters used are those defined in */etc/terminfo* (type 1 box characters) or defaulted during the initialization.

The **cbox** routine is implemented as a macro that invokes **superbox**.

The **cbox** routine returns **ERR** if the window includes the lower right corner of the display and **scrollok** is not set on.

chgat (*numchars, mode*)

wchgat (*win, numchars, mode*)

pchgat (*pane, numchars, mode*)

mvchgat (*line, col, numchars, mode*)

mvwchgat (*win, line, col, numchars, mode*)

mvpchgat (*pane, line, col, numchars, mode*)

The **chgat** routine changes the attributes of the next *numchars* characters on the window starting from the current (*line, col*) coordinates. The attributes are changed to the attributes specified by the *mode* parameter. This routine will not wrap around to the next line; however, specifying a value for the *numchars* parameter that would cause a line wrap is not an error.

The *mode* parameter is one or more of the attributes defined by the global attribute variables. More than one attribute may be specified by logically OR-ing them together. The following example changes the attributes of the next 10 characters to bold blue characters on a black background:

```
chgat (10, BOLD | F_BLUE | B_BLACK)
```


Extended Curses

The **chgat** routine returns **ERR** if the change forces scrolling and **scrollok** is not set on for the window.

clear (**)**
wclear (*win*)

The **clear** routine resets the entire **stdscr** window to blank characters. **clear** sets the current (*line*, *col*) coordinates to (0, 0).

clearok (*scr*, *boolf*)
WINDOW **scr*;

The **clearok** routine sets the clear flag for the screen specified by the *scr* parameter. If the *boolf* parameter is **TRUE**, then the screen will be cleared on the next call to **refresh** or **wrefresh**. If the *boolf* parameter is **FALSE**, then the screen will not be cleared on the next call to **refresh** or **wrefresh**. This only works on screens, and, unlike **clear**, does not alter the contents of the screen. If the *scr* parameter is **curscr**, then the next **refresh** will cause a clear-screen, even if the window passed to **refresh** is not a screen.

The **clearok** routine returns **ERR** if the window is not a full screen window.

clrtobot (**)**
wclrtobot (*win*)

The **clrtobot** routine erases the window from the current (*line*, *col*) coordinates to the bottom. **clrtobot** leaves the current (*line*, *col*) coordinates unchanged. This does not force a clear-screen sequence on the next refresh.

The **clrtobot** routine always returns the value **OK**.

clrtoeol (**)**
wclrtoeol (*win*)

The **clrtoeol** routine clears the window from the current (*line*, *col*) coordinates to the end of the current line. The current (*line*, *col*) coordinates are not changed.

The **clrtoeol** routine always returns the value **OK**.

colorend (**)**
wcolorend (*win*)

The **colorend** routine returns the terminal to **NORMAL** mode. By default, **NORMAL** is usually defined as (**F_WHITE** | **B_BLACK**).

The **colorend** routine is a macro that invokes **xstandend**.

The **colorend** routine always returns the value **OK**.

colorout (*mode*)
wcolorout (*win*, *mode*)

The **colorout** routine sets the window's current standout bit-pattern (*win*->_csbp) to the attribute specified by the *mode* parameter. Characters added to the window after such a call will have *mode* as their attribute. The *mode* parameter is constructed by logically OR-ing together attribute constants that are defined in the **cur01.h** header file that are supported by the terminal.

The **colorout** routine overrides the current setting of the window, and will work in conjunction with almost all of the routines that cause output to be placed on the window.

The **colorout** routine is a macro that invokes **xstandout**.

The **colorout** routine always returns the value **OK**.

cresetty (*boolf*)

The **cresetty** routine resets the terminal to the state saved by the last call to **csavetty**. Use this routine after the completion of a program that uses the terminal as a simple terminal. If the *boolf* parameter is **TRUE**, then the data in **curscr** is redisplayed.

crmode ()
nocrmode ()

The **crmode** routine turns off the *canonical processing* of input by the system device driver. When canonical processing is off, data is made available without waiting for a \n (new-line character). **nocrmode** enables canonical processing by the system device driver.

The **wgetch** routine, which is used for all Extended Curses input, forces the equivalent of **crmode** before requesting input if echoing is active, and reinstates the original status on exit. If you are using echo, you should issue a call to either **crmode** or **raw** to avoid multiple calls by **wgetch**.

The **crmode** routine differs from **raw** in that **crmode** has no effect on output data processing and does not disable signal processing by the device driver.

The **crmode** routine always returns the value **OK**.

csavetty (*boolf*)

The **csavetty** routine saves the current Extended Curses state so that it can later be reset by **cresetty**. Use this routine before running a program that uses the terminal as a simple terminal. If the *boolf* parameter is **TRUE**, then the following

Extended Curses

status is set before saving the terminal status: **crmode**, **noecho**, **meta**, **nonl**, and **keypad (TRUE)**.

delay

See **nodelay** on page 3-146.

delch ()

wdelch (win)

mvdelch (line, col)

mvwdelch (win, line, col)

The **delch** routine deletes the character at the current (*line*, *col*) coordinates. Each character after the deleted character on the line shifts to the left, and the last character becomes blank.

The **delch** routine always returns the value **OK**.

deleteln ()

wdeleteln (win)

The **deleteln** routine deletes the current line. Every line below the current line moves up, and the bottom line becomes blank. The current (*line*, *col*) coordinates remain unchanged.

The **deleteln** routine always returns the value **OK**.

delwin (win)

The **delwin** routine deletes the window specified by the *win* parameter. All resources used by the deleted window are freed for future use.

If a window has a subwindow allocated inside of it, the deletion of the window does not affect the subwindow even though the subwindow is invalidated. Therefore, subwindows must be deleted before the outer windows are deleted.

The **delwin** routine always returns the value **OK**.

drawbox (win, line, col, numlines, numcols)

The **drawbox** routine draws a box with the upper left corner located at the position specified by the *line* and *col* parameters. The *numlines* parameter specifies the number of rows to be used by the box, and the *numcols* parameter specifies the number of columns to be used by the box.

The characters used to draw the box are either those specified in the **terminfo** file, or those defaulted at initialization.

The **drawbox** routine returns **ERR** if part or all of the box is outside the window, or the box addresses the lower right corner of the screen and **scrollok** is not on.

#include <cur05.h>

ecactp (*pane*, *boolf*)

The **ecactp** routine specifies the active pane in a panel. The pane specified by the *pane* parameter is made the active pane if the *boolf* parameter is **TRUE**. If an active pane has been previously designated, then the border of that pane is reset to the inactive display mode, and the border of the pane specified by the *pane* parameter is set to the mode implied by the *boolf* parameter. If the *boolf* parameter is **TRUE**, then the pane is made the active pane by setting its pointer in the panel structure for the active pane.

#include <cur05.h>

ecadpn (*pane*, *win*)

The **ecadpn** routine adds the window specified by the *win* parameter to the list of windows that can be presented in the pane specified by the *pane* parameter. No visible action occurs as a result of this routine. A call to **ecaspn** must be made after **ecadpn** to change the data associated with the pane display.

The **ecadpn** routine returns **ERR** if the system is unable to allocate the storage required.

#include <cur05.h>

ecaspn (*pane*, *win*)

The **ecaspn** routine makes the window specified by the *win* parameter the current window for display in the pane specified by the *pane* parameter. A refresh call for the pane or panel is needed to cause the data to be presented on the display. The viewport associated with the pane is positioned with the top left corner of the viewport at the top left corner of the data for the window.

The **ecaspn** routine returns **ERR** if the window specified by the *win* parameter was not previously associated with this pane using **ecadpn**.

WINDOW *ecblks ()

The **ecblks** routine returns a pointer to a window that is filled with blanks. This window is intended to be used as a filler for panes that have no real content. It requires less storage than normal windows because all lines will always contain blanks.

Do not modify or delete this window.

Extended Courses

```
#include <cur05.h>
```

```
PANEL *ecbpls (numlines, numcols, firstline, firstcol, title, divdim, border, pane)  
short numlines, numcols, firstline, firstcol;  
char *title;  
char divdim, border;
```

The **ecbpls** routine builds a panel structure.

The *numlines* parameter specifies the panel size in rows.

The *numcols* parameter specifies the panel size in columns.

The *firstline* parameter specifies the panel's origin on the display's upper left corner row coordinate.

The *firstcol* parameter specifies the panel's origin on the display's upper left corner column coordinate.

The *title* parameter points to a title string. The title is shown centered in the top border. If no title is desired, this parameter should be **NULL**.

The *divdim* parameter specifies the dimension along which this panel is to be divided: either **Pdivtyv** (vertical) or **Pdivtyh** (horizontal).

The *border* parameter indicates whether or not this panel is to have a border: either **Pbordry** (yes) or **Pbordrn** (no).

The *pane* parameter points to the first pane that defines the divisions of this panel.

All parameters should be given as defined here. However, they are not checked or used until a call is made to **ecdvpl**. An application may modify values put into this structure until it calls **ecdvpl**.

Upon successful completion, a pointer to the new panel is returned. **ecbpls** returns **ERR** if there is not enough storage available.

```
#include <cur05.h>
```

```
PANE *ecbpns (numlines, numcols, ln, ld, divdim, ds, du, border, lh, lv)  
short numlines, numcols, ds;  
PANE *ln, *ld, *lh, *lv;  
char divdim, du, border;
```

The **ecbpns** routine builds a pane structure.

The *numlines* parameter specifies the number of rows in the presentation space for the pane.

The *numcols* parameter specifies the number of columns in the presentation space for the pane.

The *ln* parameter points to a neighboring pane either above or to the left.

The *ld* parameter points to the start of a chain for divisions of the pane.

The *divdim* parameter specifies the dimension of the pane along which division is to occur. This parameter is used if and only if the *ld* parameter is not **NULL**. Valid values for this parameter are **Pdivpvn** (vertical dimension) and **Pdivpnh** (horizontal dimension).

The *ds* and *du* parameters together specify the size of this pane as part of the division of a parent pane:

du **Vertical or Horizontal Size of the Pane**

Pdivszc The size is specified by the *ds* parameter.

Pdivszp The size is $ds \div 10000$ of the available space. For example, if *ds* is 5000, then the row or column size is half of the available space.

Pdivszf The pane has a “floating” size. The value of the *ds* parameter is not used.

The *border* parameter specifies whether or not this pane has a border: either **Pbordry** (yes) or **Pbordrn** (no).

The *lh* parameter points to a pane that is to scroll with this pane when the pane scrolls horizontally.

The *lv* parameter points to a pane that is to scroll with this pane when the pane scrolls vertically.

If you specify **NULL** for the *ld* parameter, or if you are not sure which value to use for *du*, then specify **Pdivszf** for the *du* parameter.

If the *ln* parameter is not **NULL**, the indicated structure is modified to point to the new structure being created in the **divs** field; the **divs** field of the new structure receives the value that was in the *ln.divs* field.

If the *lh* and the *lv* parameters are not **NULL**, they will be used to link the new structure to the specified structures and to link the specified structures to the new structure. The links thus created will form a ring that includes all panes that scroll together.

Upon successful completion, a pointer to the new pane structure is returned. **ecbpngs** returns **ERR** if a error is detected during processing.

```
#include <cur05.h>
```

```
ecdfpl (panel, boolf)
```

The **ecdfpl** routine creates the Extended Curses **WINDOW** structures needed to define the specified panel.

Extended Curses

At the time this routine is invoked, all size and location specifications of the panel and its constituent panes must be properly set. **ecdfpl** does not examine any of the division size specifications or the scroll link specifications.

The **fpane** pointer in the indicated **PANEL** structure must point to the first leaf pane for the panel, and the subsequent **nxtpn** pointers from that pane must form a loop back to the first leaf pane. (This is done by **ecdvpl**.)

A **WINDOW** structure is built for the panel specified by the *panel* parameter. This **WINDOW** will have a size that corresponds to the size of the panel. For each of the panes in the subsequent chain, a separate **WINDOW** structure is built with a size that corresponds to the specified presentation space size or the viewport size, whichever is larger.

If borders are specified for any of the panes, those borders are drawn on the **WINDOW** for the panel. All corners are checked and, if needed, proper junction characters are used to draw the corner.

The *boolf* parameter indicates whether to suppress the creation of presentation spaces for the panes. If the value is **TRUE**, then presentation spaces are not created. If **FALSE**, then presentation spaces are created.

The **ecdfpl** routine returns **ERR** if sufficient storage is not available for the **WINDOW** structures being created.

```
#include <cur05.h>
```

```
ecdppn (pane, oldwin, newwin)
```

The **ecdppn** routine adds, drops or replaces a presentation space for a pane.

First, if the *oldwin* parameter is not **NULL**, then **ecdppn** drops *oldwin* from the list of windows that are alternatives for the pane specified by the *pane* parameter. The previous association should have been established using **edadpn**. If the *oldwin* parameter is **NULL**, then no window is dropped.

Next, if the *newwin* parameter is not **NULL**, then **ecdppn** adds *newwin* as a valid pane for this window, replacing *oldwin*, if it was associated with the pane specified by the *pane* parameter. (See **ecadpn** for a better way to add a pane).

The **ecdppn** routine always returns the value **OK**.

```
#include <cur05.h>
```

```
ecdspl (panel)
```

The **ecdspl** routine releases all of the data structures associated with the panel specified by the *panel* parameter. The released data structures are returned to the free pool. The released data structures include the panel structure, all associated pane structures, any window structures associated with the panes, any auxiliary

window structures associated with the panes, and all private control structures used by Extended Curses.

#include <cur05.h>

ecdvp1 (*panel*)

The **ecdvp1** routine assigns a real size and relative position to all the panes defined for the panel specified by the *panel* parameter. All of the panes must be linked to the panel. The structure of a tree will be followed to determine the sizes for each pane.

The direction of the first set of divisions and the size of the first set of divisions is determined. This information is used to control the division algorithm. Using the size along the direction of division, first, the total space for the interior of panes is determined by counting the panes and their borders. Next, any panes with *fixed* size are given the space indicated by the **divsz** field in the pane structure. The remaining available space is then assigned to the panes that have specified a *proportional* size. Finally, any space that remains is assigned to those panes that specified a *floating* size. Once the sizes are determined, the origin for each pane relative to the panel origin is determined and entered into the PANE structure. A final pass is made over the list of panes in the current division, and, for each that is itself divided, the process is repeated.

If adjacent panes both have a border specified, the border space is shared between them.

If all of the panes have a fixed size and the total is less than the available space, there will be space that cannot be accessed by the application in the resulting structure.

If, after allocating space to the proportional panes, there is space remaining and no floating panes are in the current set, the remaining free space is allocated to the proportional panes.

The **ecdvp1** routine returns **ERR** and the structures are invalid for use by **ecdfpl** if one or more of the following occur:

- The total size specified for fixed panes exceeds the space available.
- The total fractions specified for the proportional panes exceed a total of 1.
- The number of panes exceeds the number of positions available.

#include <cur05.h>

ecflin (*pane*, *firstline*, *firstcol*, *numlines*, *numcols*, *pat*, *ch*, *bf*, *mask*)
char **pat*, **bf*, **mask*;

The **ecflin** routine inputs field data. The field data is input to the pane pointed to by the *pane* parameter. The *firstline* and the *firstcol* parameters specify the upper left corner of the field in the current window being shown in the pane. The

numcols parameter specifies the number of columns in the field. The *numlines* parameter specifies the number of rows in the field. The *bf* parameter points to a buffer into which the input data is to be stored. The buffer must be at least $\text{numlines} \times \text{numcols}$ characters long.

The *ch* parameter specifies the first character to be entered into the field. If the *ch* parameter is a null character, it is ignored.

The *pat* and *mask* parameters specify the set of characters that are to be accepted as valid input.

Input is accepted from the terminal as long as the cursor remains within the bounds of the field. Cursor movements that move the cursor outside the field is allowed and is reflected on the display. Any data keys entered are checked against the character set specified by the *pat* parameter. If the data character is acceptable, then it is echoed. If the character is not acceptable, then the **ecflin** routine returns its value.

Insert and delete keys are honored and data is shifted within the field as needed. If the field spans more than one line and insertions or deletions are made, then data that is shifted out of one line of the field is shifted into the end of the next line. Data shifted out of the field is lost. When characters are deleted, null characters are shifted into the end of the field.

The *pat* parameter points to a string that indicates the set of characters that are acceptable as valid input. The string is constructed from the following codes:

- U Uppercase letters: 'A'-'Z'.
- L Lowercase letters: 'a'-'z'.
- N Numeric characters: '0'-'9'.
- A Alphanumeric characters: 'A'-'Z', 'a'-'z', and '0'-'9'.
- B Blank or space character.
- P Printable characters: *blank*-'~' (ASCII 0x20–0x7E).
- G Graphic characters: '!'-'~' (ASCII 0x21–0x7E).
- X Hexadecimal characters: '0'-'9', 'A'-'F', and 'a'-'f'.
- C Control Characters:
 - Cursor Up, Cursor Down, Cursor Left, Cursor Right
 - Backspace
 - Back-tab (to first position of field)
 - Insert (enable or disable insert mode)
 - Delete (delete current character)
 - New-line (to left column and down one line)
- D Default characters: ASCII 0x20–0x7E, 0x80–0xFE, and controls (as defined for code C).
- Z Application-specified character set
- + Allows characters indicated by following codes.
- Does not allow characters indicated by following codes.

If the first character of *pat* is + or -, then the set of characters specified by the rest of the string is added to (+) or taken from (-) the default characters (which can also be specified with D). If the first character in this string is not + or -, then the set of characters specified by *pat* replaces the default. After the first character, the sets indicated are allowed unless preceded by a - (minus sign). For example:

"PC-L" Allows the printable and control characters, except for lowercase letters.
 "-CBN" Allows all of the default characters, except for control characters, blanks, or numeric characters.

If the *pat* string contains a Z, then the array pointed to by the *mask* parameter specifies a character validity mask. This array must be exactly 64 bytes long (512 bits), where each bit corresponds to a character code as returned by **wgetch**. If a given bit is set to 1, then the corresponding character will be accepted (for +Z) or rejected (for -Z). If a bit is set to 0, then the corresponding character's acceptance, as determined by the rest of *pat*, is not changed.

Upon successful completion, the code associated with the last input that terminated input is returned.

The **ecflin** routine returns **ERR** if one or more of the following are true:

- There is an error in the parameters.
- The *firstline* parameter is outside the window.
- The *firstcol* parameter is outside the window.
- The *numcols* parameter is too large.
- The *numlines* parameter is too large.

echo ()
noecho ()

The **echo** routine causes the terminal to echo characters to the display. If **echo** is set on, **wgetch** places all input into the data structure for the window.

The **noecho** routine turns **echo** off. If **echo** is set off, characters will not be written to the display.

#include <cur05.h>

ecpnn (pane, boolf, ch)

The **ecpnn** routine causes the pane to accept keyboard input. The pane specified by the *pane* parameter is scrolled, if necessary, to insure that the cursor is visible on the display. Keyboard input is then accepted. If the *boolf* parameter is **TRUE** and if the input character is a simple cursor movement, then the resulting cursor position is reflected on the display. Further input is then read from the terminal. If the *boolf* parameter is **FALSE**, or if the input character is not a simple cursor movement, then the value of the input character is returned.

Extended Curses

The *ch* parameter specifies the first character to be assumed from the display. If *ch* is a null character, then it is ignored.

void **ecpnmodf** (*pane*)

The **ecpnmodf** macro marks the panel that contains the pane specified by the *pane* parameter as modified. This information is used by **ecrfpl** to determine whether a panel needs to be written to the display.

#include <cur05.h>

ecrfpl (*panel*)

The **ecrfpl** routine refreshes the panel specified by the *panel* parameter. If that panel is partially obscured by other panels, then those panels are also written to the display. If the *panel* parameter is **NULL**, then all panels that have been marked as modified (with **ecpnmodf**) are written. If any panels have been removed (with **ecrmpl**), then all panels are written.

#include <cur05.h>

ecrfpn (*pane*)

The **ecrfpn** routine refreshes the pane specified by the *pane* parameter on the display. If the pane is the active pane, then the window might be scrolled to assure that the cursor is visible. If the pane is not active, then the window is not scrolled.

The **ecrfpn** routine always returns the value **OK**.

#include <cur05.h>

ecrlpl (*panel*)

The **ecrlpl** routine returns the structures associated with the panel specified by the *panel* parameter to the free storage pool. This includes all window structures associated with the panes of the panel, all Extended Curses private structures, and any added window structures. The panel and associated pane structures are not released and can be reused.

The **ecrlpl** routine always returns the value **OK**.

#include <cur05.h>

ecrmpl (*panel*)

The **ecrmpl** routine removes the panel specified by the *panel* parameter from the list of panels that are currently being displayed. If the panel is not currently in

that list, no action is taken and no error is returned. This routine should be followed by a call to **ecrfpl** to update the display.

The **ecrmpl** routine always returns the value **OK**.

```
#include <cur05.h>
```

```
ecscpn (pane, numlines, numcols)
```

The **ecscpn** routine caused the pane specified by the *pane* parameter to be scrolled over the underlying window the distance indicated by the *numcols* and the *numlines* parameters. The *numcols* parameter specifies the distance to scroll horizontally and the *numlines* parameter specifies the distance to scroll vertically. These parameters can be positive or negative and may imply a movement that positions the viewport partially or completely off the window. If such a position results from the scroll, the scroll stops after moving as far in the indicated direction as possible. Positive values move to the right or down. Negative values move to the left or up.

If there are other panes linked to the pane specified, those panes will also scroll an amount necessary to maintain the identical horizontal or vertical positioning on the respective windows. If the resulting position requires placing the viewport partially or completely off the window, the scroll request terminates at the edge of the window.

```
#include <cur05.h>
```

```
ecshpl (panel)
```

The **ecshpl** routine shows the panel specified by the *panel* parameter on the terminal.

If the specified panel is currently the top panel, no action is taken and no error is returned. If there is another top panel, the active pane in that panel is changed to the inactive state. The specified panel is placed at the top of the panel chain. This routine should be followed by a call to **ecrfpl** to update the display.

The **ecshpl** routine always returns the value **OK**.

```
#include <cur05.h>
```

```
ectitl (title, line, col)
```

```
char *title;
```

The **ectitl** routine creates or modifies the title panel. The title panel is always visible, that is, on top of any other panels. The *title* parameter points to a character string that is displayed as the new title. If *title* is **NULL**, then any existing title is removed. The *line* and *col* parameters specify the coordinates for

Extended Curses

the upper left corner of the title panel. If *firstline* is not valid, then it defaults to 1. If *firstcol* is not valid, then the title will be centered.

endwin ()

The **endwin** routine ends window routines before exiting. Ending window routines before exiting restores the terminal to the state it was before **initscr** (or **gettmode** and **setterm**) was called. **endwin** should always be called before exiting. **endwin** does not exit.

erase ()

werase (*win*)

perase (*pane*)

The **erase** routine clears the window and sets it to blanks without setting the clear flag. Similarly, **perase** erases the pane specified by the *pane* parameter. This is analogous to the **clear** routine, except that it does not cause a clear-screen sequence to be generated on a **refresh**.

flash ()

The **flash** routine displays a visual “bell” on the terminal screen if one is available. If a visual bell is not available, then **flash** toggles the terminal speaker or bell.

The **flash** routine always returns the value **OK**.

fullbox (*win, vert, hor, topl, topr, botl, botr*)

char *vert, hor, topl, topr, botl, botr;*

The **fullbox** routine puts box characters on the edges of the window. The *vert* parameter specifies the character to use for the vertical sides. The *hor* parameter specifies the character to use for the horizontal lines. The *topl* and the *topr* parameters specify the characters to use for the top left and the top right corners. The *botl* and the *botr* parameters specify the characters to use for the bottom left and the bottom right corners.

The **fullbox** routine returns **ERR** if an attempt is made to scroll when **scrollok** is not active.


```
#include <cur02.h>
```

```
int getch ( )
```

```
int wgetch (win)
```

```
int mvwgetch (line, col)
```

```
int mvwgetch (win, line, col)
```

The **getch** routine gets a character from the terminal and echoes it on the window, if necessary. If **noecho** has been set, then the window does not change. **noecho** and either **crmode** or **raw** must be set for Extended Curses to know what is actually on the terminal. If these settings are not correct, **wgetch** sets **noecho** and **crmode** and resets them to the original mode when done.

Upon completion, the character code for the data key or one of the following values is returned:

KEY_NOKEY	nodelay is active and no data is available.
KEY-xxxx	keypad is active and a control key was recognized. See the cur02.h header file for a complete list of the key codes that can be returned.
ERR	Echoing the character would cause the screen to scroll illegally.

```
#include <cur02.h>
```

```
int getstr (string)
```

```
int wgetstr (win, string)
```

```
int mvwgetstr (line, col, string)
```

```
int mvwgetstr (win, line, col, string)
```

The **getstr** routine gets a string through the window and stores it in the location pointed to by the *string* parameter. The area pointed to must be large enough to hold the string. **getstr** calls **wgetch** to get the characters a new-line character or some other control character is encountered.

Upon completion, one of the following values is returned:

OK	The input string was terminated with a new-line character.
KEY_NOKEY	nodelay is active and no data is available.
KEY-xxxx	The input string ended with a control key, and the code for this key was returned. See the cur02.h header file for a complete list of the key codes that can be returned.
ERR	The string caused the screen to scroll illegally.

```
gettmode ( )
```

The **gettmode** routine issues the needed control operation to the display device driver to access the processing flags into a fixed global area. **gettmode** is invoked by **initscr** and is not normally called directly by applications.

Extended Curses

getyx (*win*, *line*, *col*)

The **getyx** routine stores the current (*line*, *col*) coordinates of window specified by the *win* parameter into the variables *line* and *col*. Because **getyx** is a macro and not a subroutine, the names of *line* and *col* passed, not their addresses.

Upon successful completion, *line* and *col* contain the current row and column coordinates for the cursor in the specified window.

char **inch** ()

char **winch** (*win*)

char **mvinch** (*line*, *col*)

char **mvwinch** (*win*, *line*, *col*)

The **inch** routine returns the character at the current (*line*, *col*) coordinates on the specified window. No changes are made to the window.

Upon successful completion, the code for the character located at the current cursor location is returned.

WINDOW ***initscr** ()

The **initscr** routine performs screen initialization. **initscr** must be called before any of the screen routines are used. It initializes the terminal-type data, and without it, none of the Extended Curses routines can operate properly.

If standard input is not a **tty**, **initscr** sets the specifications to the terminal whose name is pointed to by **Def_term** (initially "dumb"). If the value of the **bool** global variable **My_term** is **TRUE**, **Def_term** is always used.

If standard input is a terminal, the specifications for the terminal named in the environment variable **TERM** are used. These specifications are obtained from the **terminfo** description file for that terminal.

The **initscr** routine creates the structures for **stdscr** and **curscr** and saves the pointers to those structures in global variables with the corresponding names.

Upon successful completion, a pointer to **stdscr** is returned.

insch (*ch*)

winsch (*win*, *ch*)

mvwinsch (*win*, *line*, *col*, *ch*)

mvinsch (*line*, *col*, *ch*)

The **insch** routine inserts the character specified by the *ch* parameter into the window at the current (*line*, *col*) coordinates. Each character after the inserted character shifts to the right and the last character on the line disappears.

The **insch** routine always returns the value **OK**.

insertln ()
wininsertln (win)

The **insertln** routine inserts a line above the current line. Each line below the current line is shifted down, and the bottom line disappears. The current line becomes blank and the current (*line*, *col*) coordinates remain unchanged.

The **insertln** routine always returns the value **OK**.

keypad (boolf)

The **keypad** routine turns on and off the mapping of key sequences to single integers. If the *boolf* parameter is **TRUE**, input processing is turned on. If the *boolf* parameter is **FALSE**, input processing is turned off. By default, input processing is initially turned off.

When turned on, sequences of characters from the terminal are translated into single integers that are defined in the **cur02.h** header file. The codes actually available on a given terminal are determined by the **terminfo** terminal description file.

The **keypad** routine always returns the value **OK**.

leaveok (win, boolf)

The **leaveok** routine sets the flag of type **bool** for leaving the cursor after the last change. If the *boolf* parameter is **TRUE** when the window is refreshed, the cursor is left at the last point where a change was made on the terminal, and the current (*line*, *col*) coordinates for the window specified by the *win* parameter is changed accordingly. If the (*line*, *col*) coordinates is outside the window, the coordinates are forced to (0, 0). If the *boolf* parameter is **FALSE** when the window is refreshed, the cursor is moved to the current (*line*, *col*) coordinates within the window. The *boolf* parameter is initially set to **FALSE** and will remain **FALSE** until you change it.

The **leaveok** routine always returns the value **OK**.

longname (termbuf, name)
char *termbuf, name;

The **longname** routine gets the long (full) name of the terminal described by the **terminfo** entry contained in the buffer specified by the *termbuf* parameter. The name is stored in the area pointed to by the *name* parameter.

meta ()
nometa ()

The **meta** routine prevents the stripping of the eighth bit of each keyed character.

Extended Courses

The **nometa** routine causes the eighth or most-significant bit of each keyed character to be stripped. Not all terminals support the stripping of bits.

The **meta** and **nometa** routines always return the value **OK**.

move (*line*, *col*)

wmove (*win*, *line*, *col*)

The **move** routine changes the current (*line*, *col*) coordinates of the window to the coordinates specified by the *line* and *col* parameters.

The **move** routine returns **ERR** if the destination for the cursor is outside the window or viewport.

mvcur (*line*, *col*, *newline*, *newcol*)

int *line*, *col*, *newline*, *newcol*;

The **mvcur** routine moves the terminal's cursor from the coordinates specified by the *line* and *col* parameters to the coordinates specified by the *newline* and *newcol* parameters.

It is possible to use this optimization without the benefit of the screen routines. In fact, **mvcur** should not be used with the screen routines. Use **move** and **refresh** to move the cursor position and inform the screen routines of the move.

mvwin (*win*, *line*, *col*)

The **mvwin** routine moves the position of the viewport or the subwindow specified by the *win* parameter from its current starting coordinates to the coordinates specified by the *line* and *col* parameters. The *line* parameter specifies the row on the display for the top row of the window. The *col* parameter specifies the column on the display for the first column of the window.

The **mvwin** routine returns **ERR** if a part of the window position is outside the bounds of the window on which the viewport is defined.

WINDOW *newview (*win*, *numlines*, *numcols*)

The **newview** routine creates a new window that has the number of lines specified by the *numlines* parameter and the number of columns specified by the *numcols* parameter. The new window is a viewport of the window specified by the *win* parameter and starts at the current (*line*, *col*) coordinates of the window specified by the *win* parameter. The resulting windows' initial position on the display is set to (0, 0).

The viewport window returned by **newview** is a special subwindow that is suitable for viewport scrolling. Viewport scrolling here refers to the type scrolling characteristics of full-screen editors.

Because the returned viewport window is a subwindow, any change made in either window in the area covered by the viewport window appears in both windows. Both windows actually share the relevant storage area. A viewport window cannot be scrolled using **scroll**.

Other than the exceptions noted above, viewport windows behave like subwindows.

Upon successful completion, a pointer to the control structure for the new viewport is returned.

The **newview** routine returns **ERR** if the window specified by the *win* parameter is a subwindow or a viewport, or if sufficient storage is not available for the new structures.

WINDOW *newwin (*numlines*, *numcols*, *firstline*, *firstcol*)

The **newwin** routine creates a new window that contains the number of lines specified by the *numlines* parameter and the number of columns specified by the *numcols* parameter. The new window will start at the coordinates specified by the *firstline* and the *firstcol* parameters.

If the *numlines* parameter is 0, then that dimension is set to (**LINES** - *firstline*). If the *numcols* parameter is 0, then that dimension is set to (**COLS** - *firstcol*). Therefore, to get a new window of dimensions (**LINES** × **COLS**), use:

```
newwin (0, 0, 0, 0)
```

The size specified for the window can exceed the size of the real display. In this case, a viewport or subwindow must be used to present the data from the window on the terminal.

Upon successful completion, a pointer to the new window structure is returned.

The **newwin** routine returns **ERR** if any of the parameters are invalid, or if there is insufficient storage available for the new structure.

nl (**nonl** ()

The **nl** routine sets the terminal to **nl mode**. When in **nl** mode, the system maps '\r' (return characters) to '\n' (new-line or line-feed characters). If the mapping is not done, **refresh** can do more optimization. Therefore, it is recommended that you turn **nl** mode off. **nonl** turns **nl** mode off.

The **nl** routine and **nonl** do not affect the way in which **waddch** processes new-line characters.

The **nl** and **nonl** routines always return the value **OK**.

nodelay (*boolf*)

The **nodelay** routine controls whether read requests wait for input if no keystroke is available. If the *boolf* parameter is **FALSE**, then the read routines wait for operator input. This is the default setting. If the *boolf* parameter is **TRUE**, then the read routines return immediately if no keyboard data is available.

If **nodelay** is set (**TRUE**) and if no keystroke is available from the keyboard, then **getch** returns **KEY_NOKEY**.

The **nodelay** routine always returns the value **OK**.

overlay (*win1, win2*)

The **overlay** routine overlays the window specified by the *win1* parameter on the window specified by the *win2* parameter. The contents of the window specified by the *win1* parameter, insofar as they fit, are placed on the window specified by the *win2* parameter at their starting (*line, col*) coordinates. This is done nondestructively; that is, blanks on the *win1* window leave the contents of the space on the *win2* window untouched.

The **overlay** routine moves data and flags only if the data or the display attribute is different. It uses **addch** to move data from one window to another.

The only data that is considered for moving from the *win1* window to the *win2* window is data that occupies display positions that are common to both windows.

The **overlay** routine is implemented as a macro that invokes **overput**.

The **overlay** routine returns **ERR** if the overlay addresses the lower right corner of the display and **scrollok** is **FALSE**.

overwrite (*win1, win2*)

The **overwrite** routine copies data from the window specified by the *win1* parameter to the window specified by the *win2* parameter. The contents of the *win1* window, insofar as they fit, are placed on the *win2* window at their starting (*line, col*) coordinates. This is done destructively; that is, blanks on the *win1* window become blanks on the *win2* window.

Only the data that occupies positions on the display that are common to the two windows will be moved from the *win1* window to the *win2* window.

The **overwrite** routine calls **overput** which uses **waddch** to transfer the data from window to window.

The **overwrite** routine returns **ERR** if an attempt is made to write to the lower right corner and **scrollok** is not set.


```
printw (fmt [, value, . . . ])  
wprintw (win, fmt [, value, . . . ])  
char *fmt;
```

The **printw** routine performs a **printf** on the window using the format control string specified by the *fmt* parameter and the values specified by the *value* parameters. The output to the window starts at the current (*line*, *col*) coordinates. Use the field width options of **printf** to avoid leaving things on the window from earlier calls. See “printf, fprintf, sprintf” on page 3-262 for details.

The **printw** routine returns **ERR** if it causes the screen to scroll illegally.

```
raw ()  
noraw ()
```

The **raw** routine sets the terminal to raw mode. In raw mode, canonical processing by the device driver and signal processing are turned off. The **noraw** routine turns off raw mode.

The **raw** and **noraw** routines always return the value **OK**.

```
refresh ()  
wrefresh (win)
```

The **refresh** routine synchronizes the terminal screen with the window. If the window is not a screen, then only the part of the display covered by it is updated. **refresh** checks for possible scroll errors at display time.

The **refresh** routine returns **ERR** if the change specified is in the last position of a window that includes the lower right corner of the display, or if they would cause the screen to scroll illegally. If they would cause the screen to scroll illegally, **refresh** updates whatever can be updated without causing the scroll.

```
resetty (boolf)
```

The **resetty** routine restores the terminal status flags that were previously saved by **savetty**. If the *boolf* parameter is **TRUE**, then the screen is cleared in addition to resetting the terminal. **resetty** is performed automatically by **endwin** and is not normally called directly by applications.

```
savetty ()
```

The **savetty** routine saves the current terminal status flags. **savetty** is performed automatically by **initscr** and is not normally called directly by applications.

Extended Curses

scanw (*fmt* [, *pointer*, . . .])
wscanw (*win*, *fmt* [, *pointer*, . . .])
char *fmt;

The **scanw** routine performs a **scanf** through the window using the format control string specified by the *fmt* parameter. **scanw** uses **wgetstr** to obtain the string, then invokes the internal routine for **scanf** to process the data. See “scanf, fscanf, sscanf” on page 3-281 for details.

scroll (*win*)

The **scroll** routine moves the data in the window specified by the *win* parameter up one line and inserts a new blank line at the bottom.

scrollok (*win*, *boolf*)

The **scrollok** routine sets the scroll flag for the window specified by the *win* parameter. If the *boolf* parameter is **TRUE**, then scrolling is allowed. The default setting is **FALSE**, which prevents scrolling. The scroll flag is meaningful only if the window includes the lower right corner of the display. If a character is written in that corner, then the terminal display scrolls up a line.

sel_attr (*set*)
int *set;

The **sel_attr** routine selects and prioritizes attributes from the run-time terminal. The *set* parameter points to a **NULL**-terminated integer array that contains display attribute values from the **cur01.h** header file in the order that you want them regardless of whether or not they are available.

Groups of attributes (colors and fonts) cannot be split in the array. For instance, all foreground colors specified must be in adjacent locations in the array.

The first element of a group of attributes must be the default color or font of the terminal. For example, the first foreground color specified is usually **F_WHITE**, and the first background color specified is usually **B_BLACK**.

If **sel_attr** is called after **initscr** and the display is not cleared, the attributes for the current data will not be changed until that data has been modified. This could result in unexpected attributes for some of the data on the display.

The **sel_attr** routine always returns the value **OK**.

setterm (*name*)
char **name*;

The **setterm** routine sets the terminal characteristics to be those of the terminal specified by the *name* parameter. **setterm** is called by **initscr** so you do not normally have to use it unless you wish to use just the cursor motion optimizations.

standend (**)**
wstandend (*win*)

The **standend** routine stops displaying characters in standout mode.

standout (**)**
wstandout (*win*)

The **standout** routine starts displaying characters in standout mode. Any characters added to the window are put in standout mode on the terminal if the terminal has that capability. The first available attribute as determined by **sel_attr** is used for standout. This is normally the bold attribute when the default display attribute priority is used.

The **standout** routine always returns the value **OK**.

WINDOW ***subwin** (*win*, *numlines*, *numcols*, *firstline*, *firstcol*)

The **subwin** routine creates a subwindow in the window pointed to by the *win* parameter. The subwindow has the number of lines specified by the *numlines* parameter and the number of columns specified by the *numcols* parameter. The new subwindow starts at the coordinates specified by the *firstline* and the *firstcol* parameters. Any change made to the window or the subwindow in the area covered by the subwindow is made to both windows.

The *firstline* and *firstcol* parameters are specified relative to the overall screen, not to the relative (0, 0) of the window specified by the *win* parameter.

If the *numlines* parameter is 0, then the lines dimension is set to (**LINES** - *firstline*). If the *numcols* parameter is 0, then the column dimension is set to (**COLS** - *firstcol*).

The subwindow created by **subwin** is initially aligned with the top left corner of the window. You can use **mvwin** to move the subwindow to present other areas of the window data.

Upon successful completion, a pointer to the control structure for the new subwindow is returned.

Extended Curses

The **subwin** routine returns **ERR** if the window specified by the *win* parameter already has a subwindow, or if there is insufficient storage for the new control structure.

superbox (*win, line, col, numlines, numcols, vert, hor, topl, topr, botl, botr*)
char *vert, hor, topl, topr, botl, botr*;

The **superbox** routine draws a box on the window specified by the *win* parameter. The *line* and *col* parameters specify the starting coordinates for the box. The *numlines* parameter specifies the depth of the box. The *numcols* parameter specifies the width of the box. The *vert* parameter specifies the character to use for vertical delimiting. The *hor* parameter specifies the character to use for horizontal delimiting. The *topl*, *topr*, *botl*, and *botr* parameters specify the characters to use for the top left corner, the top right corner, the bottom left corner, and the bottom right corner, respectively.

If the window specified by the *win* parameter is a **_SCROLLWIN** window, and scrolling is not allowed, the bottom right corner is not put on the window.

The **superbox** routine uses **addch** to place the characters on the window.

The **superbox** routine returns **ERR** if the defined box is outside the window, or an attempt is made to write to the lower right corner of the display when **scrollok** is off.

touchwin (*win*)

The **touchwin** routine makes it appear as if every location on the window specified by the *win* parameter has been changed. This is useful when overlapping windows are to be refreshed. A subsequent **refresh** request considers all portions of the window as potentially modified. If **touchwin** is not used, then only those positions of the window that have been addressed by an **addch** are inspected.

tstp ()

The **tstp** routine saves the current **tty** state and then put the process to sleep. When the process is restarted, the **tty** state is restored and then **wrefresh** (**curscr**) is called to redraw the screen. **initscr** sets the signal **SIGTSTP** to trap **tstp**.

The **tstp** routine always returns the value **OK**.

#include <cur04.h>

char ***unctrl** (*ch*)

The **unctrl** routine returns a string that represents the value of the *ch* parameter. Control characters become the lowercase equivalents preceded by a ^ (circumflex). Other letters are unchanged.

Upon successful completion, a pointer to the string for the parameter character is returned.

vscroll (*win*, *numlines*, *numcols*)

The **vscroll** routine scrolls the viewport specified by the *win* parameter on the window.

The *numlines* parameter specifies the direction and amount to scroll up or down. If the *numlines* parameter is positive, the viewport scrolls down the number of lines specified. If the *numlines* parameter is negative, the viewport scrolls up the number of lines specified.

The *numcols* parameter specifies the direction and amount to scroll left or right. If the *numcols* parameter is positive, the viewport scrolls to the right the number of characters specified. If the *numcols* parameter is negative, then the viewport scrolls to the left the number of characters specified.

The **vscroll** routine always scrolls as much of a requested scroll as possible. Specifying a parameter with a magnitude larger than that of the underlying window is not an error.

The **vscroll** routine calls **touchwin** if any scrolling is done.

The **vscroll** routine returns **ERR** if the window specified by the *win* parameter is not a window created by a call to **newview**.

Files

`/usr/lib/terminfo/?/*` Compiled terminal capability data base.

Related Information

In this book: “curses” on page 3-48, and “terminfo” on page 4-141.

The discussion of Extended Curses in *AIX Operating System Programming Tools and Interfaces*.

fclose, fflush

fclose, fflush

Purpose

Closes or flushes a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int fclose (stream)  
FILE *stream;
```

```
int fflush (stream)  
FILE *stream;
```

Description

The **fclose** subroutine writes buffered data to the stream specified by the *stream* parameter and then closes the stream.

The **fclose** subroutine is automatically called for all open files when the **exit** system call is invoked.

The **fflush** subroutine writes any buffered data for the stream specified by the *stream* parameter and leaves the stream open.

Return Value

Upon successful completion, both the **fclose** and the **fflush** subroutines return a value of 0. If either of these subroutines fails for any reason, then it returns the value **EOF**.

Related Information

In this book: “close” on page 2-25, “exit, _exit” on page 2-38, “fopen, freopen, fdopen” on page 3-157, “setbuf” on page 3-285, and “stdio” on page 3-297.

feof, ferror, clearerr, fileno

feof, ferror, clearerr, fileno

Purpose

Checks the status of a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int feof (stream)  
FILE *stream;
```

```
int ferror (stream)  
FILE *stream;
```

```
void clearerr (stream)  
FILE *stream;
```

```
int fileno (stream)  
FILE *stream;
```

Description

These macros inquire about the status of a stream.

The **feof** macro inquires about end-of-file. If **EOF** has previously been detected reading the input stream specified by the *stream* parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **ferror** macro inquires about input/output errors. If an I/O error has previously occurred when reading from or writing to the stream specified by the *stream* parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **clearerr** macro resets the error indicator and the **EOF** indicator to 0 for the stream specified by the *stream* parameter.

The **fileno** macro returns the integer file descriptor associated with the input pointed to by the *stream* parameter.

Note: Since these routines are implemented as macros, they cannot be declared or redeclared.

Related Information

In this book: “open” on page 2-88, “fopen, freopen, fdopen” on page 3-157, and “stdio” on page 3-297.

floor, ceil, fmod, fabs

floor, ceil, fmod, fabs

Purpose

Computes floor, ceiling, remainder, absolute value functions.

Library

Math Library (**libm.a**)

Syntax

```
#include <math.h>
```

```
double floor (x)  
double x;
```

```
double ceil (x)  
double x;
```

```
double fmod (x, y)  
double x, y;
```

```
double fabs (x)  
double x;
```

Description

The **floor** subroutine returns the largest integer (as a **double**) not greater than the x parameter.

The **ceil** subroutine returns the smallest integer not less than the x parameter.

The **fmod** subroutine returns the remainder of $x \div y$. More precisely, this value is x if the y parameter is 0. Otherwise, it is the number f with the same sign as x such that $x = iy + f$ for some integer i , and $|f| < |y|$.

The **fabs** subroutine returns the absolute value of x , $|x|$.

Related Information

In this book: “abs” on page 3-6.

fopen, freopen, fdopen

Purpose

Opens a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
FILE *fopen (path, type)  
char *path, *type;
```

```
FILE *freopen (path, type, stream)  
char *path, *type;  
FILE *stream;
```

```
FILE *fdopen (fildes, type)  
int fildes;  
char *type;
```

Description

The **fopen** subroutine opens the file named by the *path* parameter and associates a stream with it. **fopen** returns a pointer to the **FILE** structure of this stream.

The *path* parameter points to a character string that contains the name of the file to be opened.

The *type* parameter points to a character string that has one of the following values:

"r"	Open the file for reading.
"w"	Truncate or create a new file for writing.
"a"	Append (open for writing at end of file, or create for writing).
"r+"	Open for update (reading and writing).
"w+"	Truncate or create for update.
"a+"	Append (open or create for update at end of file).

The **freopen** subroutine substitutes the named file in place of the open *stream*. The original *stream* is closed whether or not the **open** succeeds. **freopen** returns a pointer to the **FILE** structure associated with *stream*. The **freopen** subroutine is typically used to attach the pre-opened *streams* associated with **stdin**, **stdout**, and **stderr** to other files.

fopen, freopen, fdopen

The **fdopen** subroutine associates a *stream* with a file descriptor obtained from an **open**, **dup**, **creat**, or **pipe** system call. These system calls open files but do not return pointers to **FILE** structures. Many of the standard I/O library subroutines require pointers to **FILE** structures. Note that the *type* of *stream* must agree with the mode of the open file.

When you open a file for update, you can perform both input and output operations on the resulting *stream*. However, an output operation cannot be directly followed by an input operation without an intervening **fseek** or **rewind**. Also, an input operation cannot be directly followed by an output operation without an intervening **fseek**, **rewind**, or an input operation that encounters the end of the file.

When you open a file for append (that is, when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. You can use **fseek** to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is ignored. All output is written at the end of the file and causes the file pointer to be repositioned to the end of the output.

If two separate processes open the same file for append, each process can write freely to the file without destroying the output being written by the other. The output from the two processes is intermixed in the order in which it is written to the file. Note that if the data is buffered, it is not actually written until it is flushed.

If the **fopen** or **freopen** subroutine fails, a **NULL** pointer is returned.

Related Information

In this book: "creat" on page 2-27, "open" on page 2-88, "fclose, fflush" on page 3-152, "fseek, rewind, ftell" on page 3-181, "setbuf" on page 3-285, and "stdio" on page 3-297.

fpfp

Purpose

Performs IEEE floating-point operations.

Library

Floating-Point Library (**libfp.a**)¹
IEEE Support Library (**libieee.a**)

Syntax

```
#include <sys/FP.h>
```

Description

The **fpfp** subroutines perform floating-point operations that support ANSI/IEEE standard 754-1985 for binary floating-point arithmetic. This section assumes that you are familiar with the details of this standard.

The **fpfp** package supplies six abstract registers, each of which can hold a single-precision (**FP_FLOAT**) or double-precision (**FP_DOUBLE**) floating-point value. It also provides a status register that controls and reflects the result of the floating-point operations.

This interface is used by compilers to implement expressions that involve floating-point values. It is not intended to be used directly by user programs. By accessing the floating-point routines through a system-defined table, programs can transparently access either software floating-point routines, or the hardware Floating-Point Accelerator, if it is installed in the system.

For greater performance, you can make a user program access the hardware Floating-Point Accelerator directly by compiling it with **fcc** or **cc -f**, instead of using the regular **cc** command. A program compiled with **fcc** or **cc -f** will run correctly only if the Floating-Point Accelerator is installed. For more details, see the discussion of the **cc** command in *AIX Operating System Commands Reference*.

¹ The **libfp.a** library contains subroutines that implement the floating-point operations in software. This library is not normally linked into user programs. Instead, the table-driven mechanism described under "C Language Interface" on page 3-160 is normally generated by language compilers.

Note: While it is possible to access the hardware registers of the Floating-Point Accelerator directly, doing so is strongly discouraged because this makes programs hardware-dependent. Programs that use the table-driven or fixed entry-point interfaces are independent not only of the characteristics of the floating-point hardware, but of whether a Floating-Point Accelerator is installed at all.

C Language Interface

Although not recommended, these subroutines can be referenced directly by C programs. They must be invoked through the external array **_fpfpf**, which contains pointers to the subroutines. This array is indexed by values of the **FPFPI** enumeration data type.

For example, the C code to read the status register is:

```
FP_STATUS fpstat;
```

```
/*((unsigned *)&fpstat) = (*(unsigned (*) ( ))_fpfpf[(int)FP_getst]) ( );
```

Note that the subroutine **_FPgetst** is invoked by indexing the **_fpfpf** array with the **FP_getst** enumeration constant.

The following example adds double values in registers 4 and 5, stores the result in register 4, and then returns the result as well:

```
FP_DOUBLE drslt;
```

```
/*((FP_DOUBLE *)&drslt) = (*(FP_DOUBLE (*)())_fpfpf[(int)FP_add])(4, 5);
```

The **FPFPI** enumeration data type is defined in the **sys/fpfp.h** header file, and it contains the following values:

FP_rdf,	FP_rdd,	FP_i2f,	FP_i2d,
FP_cpf,	FP_cpfi,	FP_cpd,	FP_cpdi,
FP_f2d,	FP_f2di,	FP_d2f,	FP_d2fi,
FP_ngf,	FP_ngfi,	FP_ngd,	FP_ngdi,
FP_abf,	FP_abfi,	FP_abd,	FP_abdi,
FP_ntf,	FP_ntfi,	FP_ntd,	FP_ntdi,
FP_rnf,	FP_rnfi,	FP_rnd,	FP_rndi,
FP_trf,	FP_trfi,	FP_trd,	FP_trdi,
FP_flf,	FP_flfi,	FP_fld,	FP fldi,
FP_cmf,	FP_cmfi,	FP_cmd,	FP_cmdi,
FP_adf,	FP_adfi,	FP_add,	FP_addi,
FP_sbf,	FP_sbfi,	FP_sbd,	FP_sbdi,
FP_mlf,	FP_mlfi,	FP_mld,	FP_mldi,

FP_dvf,	FP_dvfi,	FP_dvd,	FP_dvdi,
FP_rmf,	FP_rmfi,	FP_rmd,	FP_rmdi,
FP_sqf,	FP_sqfi,	FP_sqd,	FP_sqdi,
FP_csf,	FP_cfs,	FP_csd,	FP_cds,
FP_getst,	FP_setst,	FP_lmr,	FP_smr,
FP_tan,	FP_atan,	FP_2xm1,	FP_y12x,
FP_y1p1,	FP_d2f2,	FP_f2d2,	
FP_add3,	FP_adf3,	FP_sbd3,	FP_sbf3,
FP_m1d3,	FP_m1f3,	FP_dvd3,	FP_dvf3

The operations named **FP-xxxx** in the preceding table correspond to the subroutines named **-FPxxxx** in "Subroutines" on page 3-164.

Fixed Entry Points

In order to minimize the overhead of calling the floating-point routines that are used most frequently, an alternate interface that uses a nonstandard calling sequence is available for some of the operations. Because this is a nonstandard interface, you should not attempt to use it in programs written in high-level languages. Use this interface in assembler language programs instead.

These entry points are located at fixed locations in low memory. This fact allows programs to branch directly to the routines with a **balax** instruction instead of using the table-driven interface, avoiding four 032 Microprocessor instructions that set up the constant pool pointer and get the address of the routine indirectly.

The subroutines are the same as those described in "Subroutines" on page 3-164, except that the first parameter is passed in general-purpose register 2 of the 032 Microprocessor, and the second parameter is passed in register 3 (for a register number or a float immediate value) or in registers 3 and 4 (for a double immediate value).

The following table lists the address of each fixed entry point and the 032 Microprocessor general-purpose registers that are modified. Other registers need not be saved before calling the routine.

Operation	Entry Point	Registers Modified
_FPrdf	0x1000	2 3
_FPrdd	0x1040	2 3
_FPcpf	0x1080	2 3
_FPcpfi	0x10C0	2 3
_FPcpd	0x1100	2 3

Operation	Entry Point	Registers Modified
_FPcpdi	0x1140	2 3 4
_FPf2d	0x1180	0 2 3
_FPf2di	0x11C0	0 2 3
_FPd2f	0x1200	0 2 3
_FPd2fi	0x1240	0 2 3 4
_FPadf	0x1280	2 3
_FPadfi	0x12C0	2 3
_FPadd	0x1300	2 3
_FPaddi	0x1340	2 3 4 5
_FPsbf	0x1380	2 3
_FPsbfi	0x13C0	2 3
_FPsbd	0x1400	2 3
_FPsbdi	0x1440	2 3 4 5
_FPmlf	0x1480	2 3
_FPmlfi	0x14C0	2 3
_FPmld	0x1500	2 3
_FPmldi	0x1540	2 3 4 5
_FPdvf	0x1580	2 3
_FPdvfi	0x15C0	2 3
_FPdvd	0x1600	2 3
_FPdvdi	0x1640	2 3 4 5
_FPgetst	0x1680	2 3

Example

The following assembler subroutine adds 1.0 and 2.0 in single precision and returns the result. The subroutine can be called from a C-language program. See "Parameters" on page 3-163 about specifying whether or not a floating-point operation returns the result.

```
.set    NORESULT,0x08

.text
.globl .add12
.add12:
    stm    14,-44(1)      # Save 032 registers 14 and 15
    cal    1,-44(1)
    lrr    14,0
```

```

# Load 2.0 into float register 0; don't return the result
    cau    3,0x4000(0)    # Specify 2.0 as second parameter
    balax   0x10C0        # FP_cpfi - copy float immediate
    lis     2,0|NORESULT  # Specify float reg 0 as first parameter;
                        #    don't return the result

# Add 1.0 to float register 0; return the result
    cau    3,0x3F80(0)    # Specify 1.0 as second parameter
    balax   0x12C0        # FP_adfi - add float immediate
    lis     2,0           # Specify float reg 0 as first parameter;
                        #    DO return the result

# The result is now in 032 register 2
# Return the result in register 2 (C-language calling convention)
    lm      14,0(1)
    brx     15
    cal     1,44(1)

    .data   3
    .globl _add12
_add12     .long  .add12

```

Parameters

The following parameter declarations apply to all of the floating-point subroutines:

```

int          r1, r2, r3;
int          ival;
FP_FLOAT     fval;
FP_DOUBLE    dval;
unsigned char *dec;
FP_STATUS    status;

```

The *r1*, *r2*, and *r3* parameters are integers in the range 0 to 5, denoting one of the six abstract floating-point registers. In addition, the *r1* parameter can be logically OR-ed with the value **NORETBIT** (called the *no-result flag*), which permits greater performance by suppressing the return value from the subroutine. This allows the central processor and the Floating-Point Accelerator to operate concurrently: after issuing a request to the Floating-Point Accelerator, the central processor normally waits for it to finish the operation and return the result. If the no-result flag is set, then the central processor does not wait, but continues running while the Floating-Point Accelerator performs the

requested operation. The value returned by an operation is undefined if the no-result flag is set.

Subroutines

The following subroutines are grouped by function. The subroutine names shown are those that appear in the Floating-Point Library (**libfp.a**). See “C Language Interface” on page 3-160 for the recommended interface to these routines. The no-result flag applies to all of them, except as noted.

Load and Store Operations

FP_FLOAT **_FPrdf** (*r1*)
FP_DOUBLE **_FPrdd** (*r1*)

Returns the float or double value that is in register *r1*.

FP_FLOAT **_FPi2f** (*r1*, *ival*)
FP_DOUBLE **_FPi2d** (*r1*, *ival*)

Converts the integer value to float or double, stores the converted value into register *r1*, and returns that value.

FP_FLOAT **_FPcpf** (*r1*, *r2*)
FP_FLOAT **_FPcpfi** (*r1*, *fval*)
FP_DOUBLE **_FPcpd** (*r1*, *r2*)
FP_DOUBLE **_FPcpdi** (*r1*, *dval*)

Copies the float or double value from register *r2* or from the *val* parameter to register *r1*, and returns that value.

FP_DOUBLE **_FPf2d** (*r1*)
FP_DOUBLE **_FPf2di** (*r1*, *fval*)
FP_FLOAT **_FPd2f** (*r1*)
FP_FLOAT **_FPd2fi** (*r1*, *dval*)

Converts the float or double value in register *r1* or in the *val* parameter to double or float, stores it into register *r1*, and returns that value.

FP_DOUBLE **_FPf2d2** (*r1*, *r2*)
FP_FLOAT **_FPd2f2** (*r1*, *r2*)

Converts the float or double value in register *r2* to double or float, stores it into register *r1*, and returns that value. The value in register *r2* remains unmodified.

Multiple-Register Load and Store Operations

_FPImr Loads multiple floating-point registers with values stored in memory.

_FPsmr Stores multiple floating-point registers into a buffer in memory.

These subroutines require a nonstandard calling sequence that can be performed in assembler language programs, but not using high-level languages.

Both subroutines are passed the following parameters in 032 Microprocessor general-purpose registers:

Reg	Contents
12	Specifies the address of the save area.
11	The low-order 16 bits contain a bit mask that specifies the floating-point registers to load or store. Single-precision fpfp registers 0 through 5 are selected by setting the even bits 0, 2, 4, 6, 8, and 10, where bit 0 is the least significant bit of the mask. The double-precision fpfp registers 0 through 5 are selected by pairs of bits: 0-1, 2-3, 4-5, 6-7, 8-9, and 10-11. For example, setting bits 4 and 5 selects double-precision fpfp register 2. This register mapping is shown in Figure 3-1 on page 3-175.

Both operations modify 032 Microprocessor registers 0, 11, 12, and 13.

Unary Operations

FP_FLOAT **_FPngf** (*r1*)

FP_FLOAT **_FPngfi** (*r1*, *fval*)

FP_DOUBLE **_FPngd** (*r1*)

FP_DOUBLE **_FPngdi** (*r1*, *dval*)

Negates the float or double value in register *r1* or in the *val* parameter, stores it into register *r1*, and returns that value.

FP_FLOAT **_FPabf** (*r1*)

FP_FLOAT **_FPabfi** (*r1*, *fval*)

FP_DOUBLE **_FPabd** (*r1*)

FP_DOUBLE **_FPabdi** (*r1*, *dval*)

Makes the float or double value in *r1* or in the *val* parameter positive without changing its magnitude, stores it into register *r1*, and returns that value.

FP_FLOAT **_FPntf** (*r1*)
FP_FLOAT **_FPntfi** (*r1*, *fval*)
FP_DOUBLE **_FPntd** (*r1*)
FP_DOUBLE **_FPntdi** (*r1*, *dval*)

Rounds the float or double value in register *r1* or in the *val* parameter to an integer in floating-point format, stores the result into register *r1*, and returns that value.

int **_FPPrnf** (*r1*)
int **_FPPrnfi** (*r1*, *fval*)
int **_FPPrnd** (*r1*)
int **_FPPrndi** (*r1*, *dval*)

Copies the *val* parameter, if given, to register *r1*, then returns the integer value that is nearest to the float or double value in register *r1*. Note that the value in register *r1* is not converted to an integer. The no-result flag does not apply to this operation. (See "Parameters" on page 3-163 for an explanation of the no-result flag.)

int **_FPPtrf** (*r1*)
int **_FPPtrfi** (*r1*, *fval*)
int **_FPPtrd** (*r1*)
int **_FPPtrdi** (*r1*, *dval*)

Copies the *val* parameter, if given, to register *r1*, then returns the integer part of the float or double value in register *r1*. Note that the value in register *r1* is not converted to an integer. The no-result flag does not apply to this operation. (See "Parameters" on page 3-163 for an explanation of the no-result flag.)

int **_FPflf** (*r1*)
int **_FPflfi** (*r1*, *fval*)
int **_FPfld** (*r1*)
int **_FPfldi** (*r1*, *dval*)

Copies the *val* parameter, if given, to register *r1*, then returns the largest integer less than or equal to the float or double value in register *r1*. Note that the value in register *r1* is not converted to an integer. The no-result flag does not apply to this operation. (See "Parameters" on page 3-163 for an explanation of the no-result flag.)

Comparison Operations

```
int  _FPcmf (r1, r2)
int  _FPcmfi (r1, fval)
int  _FPcmd (r1, r2)
int  _FPcmdi (r1, dval)
```

Compares the float or double value in register *r1* to the float or double value in register *r2* or in the *val* parameter. The value returned is **LESSTHAN**, **EQUAL**, **GREATER** or **MININT**, depending on whether the value in register *r1* is less than, equal to, greater than, or unordered with the other value, respectively. In addition, the 032 Microprocessor test bit is set to 1 if the comparison is unordered, and it is set to 0 otherwise.

If the *r1* parameter is logically OR-ed with the value **ExceptOnUnordered**, and if the operands are unordered, then an invalid operation exception occurs. (See "Status Register Operations" on page 3-171 for details about floating-point exceptions.)

The no-result flag does not apply to these comparison operations. (See "Parameters" on page 3-163 for an explanation of the no-result flag.)

Simple Arithmetic Operations

```
FP_FLOAT  _FPadf (r1, r2)
FP_FLOAT  _FPadfi (r1, fval)
FP_DOUBLE _FPadd (r1, r2)
FP_DOUBLE _FPaddi (r1, dval)
```

Adds the float or double value in register *r2* or in the *val* parameter to the float or double value in register *r1*, stores the result into register *r1*, and then returns the result.

```
FP_FLOAT  _FPsbf (r1, r2)
FP_FLOAT  _FPsbfi (r1, fval)
FP_DOUBLE _FPsbd (r1, r2)
FP_DOUBLE _FPsbdi (r1, dval)
```

Subtracts the float or double value in register *r2* or in the *val* parameter from the float or double value in register *r1*, stores the result into register *r1*, and then returns the result.

FP_FLOAT **_FPm1f** (*r1*, *r2*)
FP_FLOAT **_FPm1fi** (*r1*, *fval*)
FP_DOUBLE **_FPm1d** (*r1*, *r2*)
FP_DOUBLE **_FPm1di** (*r1*, *dval*)

Multiplies the float or double value in register *r1* by the float or double value in register *r2* or in the *val* parameter, stores the result into register *r1*, and then returns the result.

FP_FLOAT **_FPdvf** (*r1*, *r2*)
FP_FLOAT **_FPdvfi** (*r1*, *fval*)
FP_DOUBLE **_FPdvd** (*r1*, *r2*)
FP_DOUBLE **_FPdvdi** (*r1*, *dval*)

Divides the float or double value in register *r1* by the float or double value in register *r2* or in the *val* parameter, stores the result into register *r1*, and then returns the result.

FP_FLOAT **_FPrmf** (*r1*, *r2*)
FP_FLOAT **_FPrmfi** (*r1*, *fval*)
FP_DOUBLE **_FPrmd** (*r1*, *r2*)
FP_DOUBLE **_FPrmdi** (*r1*, *dval*)

Computes the remainder of the float or double value in register *r1* when divided by the float or double value in register *r2* or in the *val* parameter, stores the result into register *r1*, and then returns the result.

FP_FLOAT **_FPsqf** (*r1*)
FP_FLOAT **_FPsqfi** (*r1*, *fval*)
FP_DOUBLE **_FPsqd** (*r1*)
FP_DOUBLE **_FPsqdi** (*r1*, *dval*)

Computes the square root of the float or double value in *r1* or in the *val* parameter, stores the result into register *r1* and then returns the result.

Three-Operand Arithmetic Operations

FP_FLOAT **_FPadf3** (*r1*, *r2*, *r3*)
FP_DOUBLE **_FPadd3** (*r1*, *r2*, *r3*)

Adds the float or double values in registers *r2* and *r3*, stores the result into register *r1*, and then returns the result.

FP_FLOAT **_FPsbf3** (*r1*, *r2*, *r3*)
FP_DOUBLE **_FPsbd3** (*r1*, *r2*, *r3*)

Subtracts the float or double value in register *r3* from the value in register *r2*, stores the result into register *r1*, and then returns the result.

FP_FLOAT **_FPmlf3** (*r1*, *r2*, *r3*)
FP_DOUBLE **_FPmld3** (*r1*, *r2*, *r3*)

Multiplies the float or double values in registers *r2* and *r3*, stores the result into register *r1*, and then returns the result.

FP_FLOAT **_FPdvf3** (*r1*, *r2*, *r3*)
FP_DOUBLE **_FPdvd3** (*r1*, *r2*, *r3*)

Divides the float or double value in register *r2* by the value in register *r3*, stores the result into register *r1*, and then returns the result.

Packed Decimal Conversion Operations

FP_FLOAT **_FPcsf** (*r1*, *dec*)
FP_DOUBLE **_FPcsd** (*r1*, *dec*)

Converts the packed decimal string pointed to by the *dec* parameter to float or double, stores the result into register *r1*, and then returns the result. The *dec* parameter points to a 10-byte packed decimal number, the first byte of which is 0 (for a positive value) or 0x80 (for a negative value). The following 9 bytes containing 2 decimal digits each; the left half-byte of each byte contains the more significant digit.

void **_FPcfs** (*r1*, *dec*)
void **_FPcds** (*r1*, *dec*)

Converts the float or double value in register *r1* to packed decimal format as described for **_FPcsf** and **_FPcsd**. In the special cases of $+\infty$, $-\infty$, and NaN, the first byte is set to 0x7F, 0xFF, or 0x0F, respectively.

Basic Transcendental Operations

These operations are the core calculations for other logarithmic, hyperbolic, and trigonometric functions, and they are frequently used as partial steps in calculating the result desired by the user. They are not implemented by the Floating-Point Accelerator, but by software routines only.

FP-DOUBLE -FPtan (*r1*, *r2*)

Computes the partial tangent of an angle measured in radians. Register *r1* contains the angle, which must be a double value in the range $0 < r1 \leq \pi \div 4$. If the value in register *r1* is outside this range, then an invalid operation exception occurs, and no operation is performed. The result of this operation consists of 2 double values. The sine of the angle is stored into register *r1* and is returned as the value of the subroutine. The cosine is stored into register *r2*. To compute the tangent, divide the sine by the cosine.

FP-DOUBLE -FPatan (*r1*, *r2*)

Computes the arctangent of $y \div x$. This operation requires a double value in both registers *r1* and *r2*. These arguments must be ordered such that the value in *r1* (*x*) is greater than the value in *r2* (*y*), which is greater than 0. If the values are not ordered in this manner, then an invalid operation exception occurs, and no operation is performed. The result is stored into register *r1* and is returned as the value of the subroutine.

FP-DOUBLE -FP2xm1 (*r1*)

Computes the value of $2^x - 1$. Register *r1* contains the exponent (*x*), which must be a double value in the range $0 \leq r1 \leq 0.5$. If the value in register *r1* is not in this range, then an invalid operation exception occurs, and no operation is performed. The result of this operation is stored into register *r1* and is returned as the value of the subroutine.

FP-DOUBLE -FPyl2x (*r1*, *r2*)

Computes the value of $y \cdot \log_2 x$. This operation requires a double value in both registers *r1* and *r2*. Register *r1* contains the exponent (*x*), which must be a double value in the range $0 < r1 < +\infty$. If the value in register *r1* is outside this range, then an invalid operation exception occurs, and no operation is performed. Register *r2* contains the multiplier (*y*), which can be any valid double value. The result of this operation is stored into register *r1* and is returned as the value of the subroutine.

FP-DOUBLE -FPylp1 (*r1*, *r2*)

Computes the value of $y \cdot \log_2 (x+1)$. This operation requires a double value in both registers *r1* and *r2*. Register *r1* contains *x*, which must be a double value in the range:

$$0 < r1 < \left[1 - \frac{\sqrt{2}}{2} \right]$$

If the value in register *r1* is outside this range, then an invalid operation exception occurs, and no operation is performed. Register *r2* contains the multiplier (*y*), which can be any valid double value. The result of the operation is stored into register *r1* and is returned as the value of the subroutine. This subroutine provides more accurate results than **-FPyl2x** for *x* values close to 1.

Status Register Operations

The floating-point status is kept in a **FP-STATUS** structure, which is defined with a **typedef** statement in the **sys/fpfp.h** header file, and which contains the following members:

unsigned kill	: 1	Signal SIGFPE on an exception
unsigned xcp_flag	: 1	An exception occurred
unsigned io_flag	: 1	An invalid operation occurred
unsigned io_xpt	: 1	Enable signal on invalid operation
unsigned dz_flag	: 1	A division by 0 occurred
unsigned dz_xpt	: 1	Enable signal on division by 0
unsigned of_flag	: 1	An overflow occurred
unsigned of_xpt	: 1	Enable signal on overflow
unsigned uf_flag	: 1	An underflow occurred
unsigned uf_xpt	: 1	Enable signal on underflow
unsigned cmp_rslt	: 2	Comparison result
unsigned rnd_mode	: 2	Rounding mode
unsigned ir_flag	: 1	An inexact result occurred
unsigned ir_xpt	: 1	Enable signal on inexact result

When set, the **kill** field enables exception trapping. The **io_xpt**, **dz_xpt**, **of_xpt**, **ir_xpt**, and **uf_xpt** fields enable trapping for each of the five exception types. Thus, for an exception to be trapped, both **kill** and the **xx_xpt** field for that exception must be set. See "Exception Handling" on page 3-172 for detailed information about exceptions and traps.

When an exception occurs for which trapping is not enabled, the fields **io_flag**, **dz_flag**, **of_flag**, **ir_flag**, and **uf_flag** are set to indicate the exception or exceptions that occurred. It also sets **xcp_flag** to indicate that an exception occurred. The **xcp_flag** and **xx_flag** fields are not automatically reset to 0; this must be done explicitly with **-FPsetst**.

The **cmp_rslt** field of the status word specifies the result of a comparison operation, and it is set to one of the following values:

0	Less than
1	Equal
2	Greater than
3	Unordered.

The **rnd_mode** field of the status word specifies the rounding mode, and it can be set to one of the following values:

FP_NEAR	Rounds to the nearest value. This is the default.
FP_ZERO	Rounds toward 0.
FP_UP	Rounds toward $+\infty$.
FP_DOWN	Rounds toward $-\infty$.

FP_STATUS **_FPgetst ()**

Returns the floating-point status word reflecting the results of the most recent floating-point operation.

void **_FPsetst (status)**

Sets the floating-point status word to the value provided by the *status* parameter.

Exception Handling

When an error is detected during a floating-point operation, an **exception** occurs. Each exception is one of five types, as specified by the ANSI/IEEE standard: invalid operation, overflow, underflow, division by zero, and inexact result.

The standard also defines a **default result** that is returned as the value of the operation that caused the exception. Alternatively, you can specify the action to be taken when an exception occurs. Performing a user-defined action when an exception occurs is called **exception trapping**, and a user-defined trap routine is called a **trap handler**. However, the default result is sufficient for most applications, and specifying a trap handler is usually not necessary.

When an exception occurs, the floating-point operation checks the trap-enabling fields of the floating-point status word. (See “Status Register Operations” on page 3-171 for details about the trap-enabling fields.) If either the **kill** field or the **xx_xpt** field for the exception type is not set, then the exception is not trapped and the following default actions take place:

- The **xcp_flag** field is set to indicate that an exception occurred.
- The appropriate **xx_flag** field is set to indicate the exception type.
- The default result specified by the ANSI/IEEE standard is returned to the application program.

If both the **kill** field and the **xx_xpt** field are set, then trapping is enabled for the given exception type. In this case, the **SIGFPE** signal is sent to the process. (AIX signals are discussed in “signal” on page 2-139.) You can write your own signal handler, which must determine the exception that occurred and take appropriate action, or you can use the facility provided by the **ieeetrap** subroutine.

Note: If the status word setting enables the trapping of one or more exception types and no signal handler is specified for **SIGFPE** (either a user-supplied handler or the one

installed by **ieeetrapp**), then the default action (**SIG_DFL**) is taken for the signal, which terminates the process.

The **ieeetrapp** subroutine, which is located in the IEEE Support Library (**libieee.a**), allows you to specify a trap handler for each of the five exception types. **ieeetrapp** functions like the **signal** system call and has the following syntax:

```
int (*ieeetrapp (exceptiontype, action)) ( )
int exceptiontype;
int (*action) ( );
```

The *exceptiontype* parameter identifies the type of exception and is one of the following values:

FP_INV_OPER	Invalid operation
FP_OVERFLOW	Overflow
FP_UNDERFLOW	Underflow
FP_DIVIDE	Division by zero
FP_INEXACT	Inexact result.

The *action* parameter is one of the following:

SIG_IGN	Sets the status word to disable trapping of the specified exception type. This is the default action for a given exception type if an action is not specified for it.
SIG_DFL	Causes the process to be terminated upon an exception of the specified type. The process is terminated by setting the action for the SIGFPE signal to SIG_DFL and sending SIGFPE to the process. This allows the parent process to determine that the process terminated due to a floating-point exception.
<i>handler</i>	Causes the subroutine with the address given by <i>handler</i> to be called when an exception of the specified type occurs.

Warning: Unpredictable results can occur if you use **ieeetrapp** to disable trapping of a given exception type (**SIG_IGN**) and then enable trapping by setting the status word. Instead, use **ieeetrapp** to set a trap handler for the exception type.

The **ieeetrapp** subroutine establishes a signal handler for **SIGFPE** that selects one of five actions based on the exception type. It also sets *action* as the action to be taken when an *exceptiontype* exception occurs.

Upon successful completion, the **ieeetrapp** subroutine returns the previous value of *action* for the specified *exceptiontype*. If an error occurs, then the value **BADSIG** is returned.

If the *action* parameter to **ieeetramp** specifies a *handler* subroutine, then an exception of the specified type causes the *handler* subroutine to be called with the following syntax:

```
#include <sys/FP.h>
#include <signal.h>

handler (sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

The parameters are the same as those passed to the signal handler. (See “sigvec” on page 2-150 for a detailed description.) Note that the value of the *sig* parameter is always **SIG_FPE**.

The **sigcontext** structure, which is defined in the **sys/signal.h** header file, contains a member named **fpvmp**, which points to a **fpvmach** structure. The **fpvmach** structure describes the state of the virtual floating-point machine at the time of the exception. This structure is defined in the **ieeetramp.h** header file, and it contains the following members:

fpreg	fpreg[NDPFREGS]	Set of floating-point registers
FP_STATUS	statusreg	Status register at time of exception
fptrap	fptrap	Specific information about the exception

Each **fpreg** structure in the array represents a double-precision value, two single-precision values, or two long unsigned integers. The **FP_STATUS** structure is discussed in “Status Register Operations” on page 3-171. The **fptrap** structure, also defined in **ieeetramp.h**, contains the following members:

fptrapinfo	fptrapinfo
fpreg	designated_result

The **fptrapinfo** structure contains the following members:

unsigned	operation	:	8	Operation causing the exception
unsigned	src	:	4	Source register number
unsigned	dest	:	4	Destination register number
unsigned	except_flags	:	5	Exceptions that occurred
unsigned	except_type	:	3	Type of exception

The **operation** field contains one of the values defined by the **FPFPI** enumeration data type, identifying the operation that caused the exception. See “C Language Interface” on page 3-160 for details about **FPFPI**.

The **src** and **dest** fields specify the operand registers, but not by their abstract **fpfp** register numbers. The operand registers given in **src** and **dest** are numbered from 0 to 13 for single-precision operands. Double-precision operands occupy a pair of registers identified by the even number of the pair. For example, 4 identifies the double-precision value occupying registers 4 and 5, and it corresponds to the abstract **fpfp** register 2. This register mapping is shown in the following figure:

float fpfp register number	Register number in src or dest	double fpfp register number
0	{ 0 1 }	0
1	{ 2 3 }	1
2	{ 4 5 }	2
3	{ 6 7 }	3
4	{ 8 9 }	4
5	{ 10 11 }	5
6 *	{ 12 13 }	6 *

↑
Bit set in mask for
_FPImr or **_FPsmr**

* **fpfp** register 6 is a scratch register that is reserved for use by the AIX kernel. User programs must not modify this register.

Figure 3-1. fpfp Register Mapping

The **except_flags** field indicates the exception or exceptions that resulted from the operation. This value is constructed by logically OR-ing together one or more of **EM_INV_OPER**, **EM_OVERFLOW**, **EM_UNDERFLOW**, **EM_DIVIDE**, and **EM_INEXACT**.

The **except_type** field identifies the exception that caused the trap to be taken. This is the highest-priority exception that occurred for which trapping is enabled. The value is one of **FP_INV_OPER**, **FP_OVERFLOW**, **FP_UNDERFLOW**, **FP_DIVIDE**, or **FP_INEXACT**.

On an inexact, overflow, or underflow exception, the **designated_result** in the **fptrap** structure contains the *designated result*, as defined by the ANSI/IEEE standard. It is

properly rounded to the precision of the destination of the operation. In the case of overflow or underflow, this result is also scaled.

Warning: When an exception is trapped, the status register is not automatically set to reflect the exception, nor is a result stored in the destination register. The trap handler must do this or the results will be unpredictable.

To set the status register and result value, set the appropriate members of the **fpvmach** structure. That is, set the status by modifying the bit fields of *scp->fpvmp->statusreg*, and store the result into *scp->fpvmp->fpreg[dest]*. (See the **sys/signal.h**, **ieeetrap.h**, and **sys/fpfp.h** header files for the complete definitions of these structures.)

If the floating-point operation being performed returns an integer value (**_FPnxx**, **_FPtrxx**, or **_FPflxx**), then store the result in the destination register as an integer value in double-precision floating-point format.

Related Information

The description of the Floating-Point Accelerator in *Hardware Technical Reference*.

The discussion of Floating-Point Services in *Virtual Resource Manager Technical Reference*.

Assembler Language Reference.

The **cc** command in *AIX Operating System Commands Reference*.

fread, fwrite

Purpose

Performs binary input/output.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int fread ((char *) ptr, sizeof (*ptr), nitems, stream)
int nitems;
FILE *stream;
```

```
int fwrite ((char *) ptr, sizeof (*ptr), nitems, stream)
int size, nitems;
FILE *stream;
```

Description

The **fread** subroutine copies *nitems* items of data from the input *stream* into an array beginning at the location pointed to by the *ptr* parameter. Each data item has the type **ptr*.

The **fread** subroutine stops copying bytes if an end-of-file or error condition is encountered while reading from the input specified by the *stream* parameter, or when the number of data items specified by the *nitems* parameter have been copied. **fread** leaves the file pointer of *stream*, if defined, pointing to the byte following the last byte read, if there is one. The **fread** subroutine does not change the contents of *stream*.

The **fwrite** subroutine appends *nitems* items of data of the type **ptr* from the array pointed to by the *ptr* parameter to the output *stream*.

The **fwrite** subroutine stops writing bytes if an error condition is encountered on *stream*, or when the number of items of data specified by the *nitems* parameter have been written. The **fwrite** subroutine does not change the contents of the array pointed to by the *ptr* parameter.

Return Value

The **fread** and **fwrite** subroutines return the number of items actually read or written. If the *nitems* parameter is negative or 0, no characters are read or written, and a value of 0 is returned.

Related Information

In this book: “read, readx” on page 2-104, “write, writex” on page 2-178, “fopen, freopen, fdopen” on page 3-157, “getc, fgetc, getchar, getw” on page 3-189, “gets, fgets” on page 3-205, “printf, fprintf, sprintf” on page 3-262, “putc, putchar, fputc, putw” on page 3-266, “puts, fputs” on page 3-269, “scanf, fscanf, sscanf” on page 3-281, and “stdio” on page 3-297.

frexp, ldexp, modf

Purpose

Manipulates parts of floating-point numbers.

Library

Standard C Library (**libc.a**)

Syntax

```
double frexp (value, eptr)  
double value;  
int *eptr;
```

```
double ldexp (mant, exp)  
double mant;  
int exp;
```

```
double modf (value, iptr)  
double value, *iptr;
```

Description

Every nonzero number can be written uniquely as $x \times 2^n$, where the mantissa (fraction), x , is in the range $0.5 \leq |x| < 1.0$, and the exponent, n , is an integer. The internal representation of floating-point numbers uses this fact, storing a mantissa part and an exponent part.

The **frexp** subroutine returns the mantissa of *value* parameter and stores the exponent in the location pointed to by the *eptr* parameter.

The **ldexp** subroutine returns the quantity $\textit{mant} \times 2^{\textit{exp}}$.

The **modf** subroutine returns the signed fractional part of the *value* parameter and stores the integral part in the location pointed to by the *iptr* parameter.

If the **ldexp** subroutine overflows, it returns **HUGE** sets **errno** to **ERANGE**.

Related Information

In this book: “sgetl, sputl” on page 3-289.

fseek, rewind, ftell

Purpose

Repositions the file pointer of a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int fseek (stream, offset, whence)  
FILE *stream;  
long offset;  
int whence;
```

```
void rewind (stream)  
FILE *stream;
```

```
long ftell (stream)  
FILE *stream;
```

Description

The **fseek** subroutine sets the position of the next input or output operation on the I/O stream specified by the *stream* parameter. The position of the next operation is determined by the *offset* parameter, which can be either positive or negative.

The **fseek** subroutine sets the file pointer associated with the specified *stream* as follows:

- If the *whence* parameter is 0, the pointer is set to the value of the *offset* parameter.
- If the *whence* parameter is 1, the pointer is set to its current location plus the value of the *offset* parameter.
- If the *whence* parameter is 2, the pointer is set to the size of the file plus the value of the *offset* parameter.

The **fseek** subroutine fails if attempted on a file that has not been opened using **fopen**. In particular, **fseek** cannot be used on a terminal, or on a file opened with **popen**.

Upon successful completion, **fseek** returns a value of 0. If **fseek** fails, a nonzero value is returned.

The **rewind** subroutine is equivalent to **fseek** (*stream*, (long) 0, 0), except that it does not return a value.

fseek, rewind, ftell

The **fseek** and **rewind** subroutines undo any effects of the **ungetc** subroutine.

After an **fseek** or a **rewind**, the next operation on a file opened for update can be either input or output.

The **ftell** subroutine returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

Related Information

In this book: “lseek” on page 2-65, “fopen, freopen, fdopen” on page 3-157, and “stdio” on page 3-297.

ftok

Purpose

Generates a standard interprocess communication key.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>  
#include <sys/ipc.h>
```

```
key_t ftok (path, id)  
char *path;  
char id;
```

Description

The **ftok** subroutine returns a key, based on the *path* and *id* parameters, to be used to obtain interprocess communication identifiers. The *path* parameter must be the path name of an existing file that is accessible to the process. The *id* parameter must be a character that uniquely identifies a project. **ftok** returns the same key for linked files if called with the same *id* parameter. Different keys are returned for the same file if different *id* parameters are used.

All interprocess communication facilities require you to supply a key to the **msgget**, **semget**, and **shmget** system calls in order to obtain interprocess communication identifiers. The **ftok** subroutine provides one method of creating keys, but many others are possible. Another way to do this, for example, is to use the project ID as the most significant byte of the key, and to use the remaining portion as a sequence number.

ftok

Warning: It is important for each installation to define standards for forming keys. If some standard is not adhered to, then it is possible for unrelated processes to interfere with each other's operation.

Warning: If the *path* parameter of the **ftok** subroutine names a file that has been removed while keys still refer it, then the **ftok** subroutine returns an error. If that file is then recreated, the **ftok** subroutine will probably return a different key than the original one.

Return Value

Upon successful completion, the **ftok** subroutine returns a key that can be passed to the **msgget**, **semget**, or **shmget** system call. The **ftok** subroutine returns (**key_t**) -1 if one or more of the following are true:

- The file named by the *path* parameter does not exist.
- The file named by the *path* parameter is not accessible to the process.
- The *id* parameter is 0 ('0').

Related Information

In this book: “msgget” on page 2-74, “semget” on page 2-113, and “shmget” on page 2-134.

ftw

Purpose

Walks a file tree.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;
```

Description

The **ftw** subroutine recursively searches the directory hierarchy that descends from the directory specified by the *path* parameter.

For each file in the hierarchy, the **ftw** subroutine calls the function specified by the *fn* parameter, passes it a pointer to a null-terminated character string containing the name of the file, a pointer to a **stat** structure containing information about the file, and an integer. (For information about the **stat** structure, see “stat.h” on page 5-61.)

The integer passed to *fn* identifies the file type, and it has one of the following values:

FTW_F	Regular file
FTW_D	Directory
FTW_DNR	Directory that cannot be read
FTW_NS	A file for which stat could not be executed successfully.

If the integer is **FTW_DNR**, then the files and subdirectories contained in that directory are not processed.

If the integer is **FTW_NS**, then the **stat** structure contents are meaningless. An example of an file that causes **FTW_NS** to be passed to *fn* is a file in a directory for which you have read permission but not execute (search) permission.

The **ftw** subroutine finishes processing a directory before processing any of its files or subdirectories.

The **ftw** subroutine continues the search until the directory hierarchy specified by the *path* parameter is completed, an invocation of the function specified by the *fn* parameter returns a nonzero value, or an error is detected within **ftw**, such as an I/O error.

If the directory hierarchy is completed, the **ftw** subroutine returns a value of 0. If the function specified by the *fn* parameter returns a nonzero value, **ftw** stops its search and returns the value that was returned by the function. If the **ftw** subroutine detects an error, a value of -1 is returned and **errno** is set to indicate the error.

The **ftw** subroutine uses one file descriptor for each level in the tree. The *depth* parameter specifies the maximum number of file descriptors to be used. In general, the **ftw** subroutine runs faster if the value of the *depth* parameter is at least as large as the number of levels in the tree. However, the *depth* parameter must not be greater than the number of file descriptors currently available for use. If the value of the *depth* parameter is 0 or negative, the effect is the same as if it were 1.

Because the **ftw** subroutine is recursive, it is possible for it to terminate with a memory fault due to stack overflow when applied to very deep file structures.

The **ftw** subroutine uses the **malloc** subroutine to allocate dynamic storage during its operation. If **ftw** is terminated prior to its completion, such as by **longjmp** being executed by the function specified by the *fn* parameter or by an interrupt routine, then **ftw** cannot free that storage. The storage remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the *fn* parameter return a nonzero value the next time it is called.

Related Information

In this book: “signal” on page 2-139, “stat.h” on page 5-61, “malloc, free, realloc, calloc” on page 3-224, and “setjmp, longjmp” on page 3-287.

gamma

Purpose

Computes the logarithm of the gamma function.

Library

Math Library (**libm.a**)

Syntax

```
#include <math.h>
```

```
extern int signgam;
```

```
double gamma (x)  
double x;
```

Description

The **gamma** subroutine returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as:

$$\int_0^{\infty} e^{-t} t^{x-1} dt$$

The sign of $\Gamma(x)$ is stored in the external integer variable **signgam**. The x parameter cannot be a nonpositive integer.

If the x parameter is a nonpositive integer, **gamma** returns **HUGE**, sets **errno** to **EDOM**, and writes a **DOMAIN** error message to standard error.

If the correct value overflows, **gamma** returns **HUGE** and sets **errno** to **ERANGE**.

You can change the error handling procedures with the **matherr** subroutine.

gamma

Examples

The following C program fragment calculates $\Gamma(x)$ and stores the result in y:

```
errno = 0;
y = gamma(x);
if (errno == 0)
    y = signgam * exp(y);
else
    perror("Error in gamma function");
```

Related Information

In this book: “exp, log, log10, pow, sqrt” on page 3-121 and “matherr” on page 3-226.

getc, fgetc, getchar, getw

Purpose

Gets a character or word from an input stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int getc (stream)
FILE *stream;
```

```
int fgetc (stream)
FILE *stream;
```

```
int getchar ( )
```

```
int getw (stream)
FILE *stream;
```

Description

The **getc** macro returns the next character (byte) from the input specified by the *stream* parameter and moves the file pointer, if defined, ahead one character in *stream*. **getc** is a macro and cannot be used where a subroutine is necessary; for example, a subroutine pointer cannot point to it.

Because it is implemented as a macro, **getc** does not work correctly with a *stream* parameter that has side effects. In particular, the following does not work:

```
getc(*f++)
```

In cases like this, use the **fgetc** subroutine instead.

The **fgetc** subroutine performs the same function as **getc**, but **fgetc** is a genuine subroutine, not a macro. The **fgetc** subroutine runs more slowly than **getc**, but takes less space.

The **getchar** macro returns the next character from the standard input stream, **stdin**. Note that **getchar** is also a macro.

The **getw** subroutine returns the next word (**int**) from the input specified by the *stream* parameter and increments the associated file pointer, if defined, to point to the next word. The size of a word varies from one machine architecture to another. The **getw** subroutine

getc, fgetc, getchar, getw

returns the constant **EOF** at end-of-file or when an error occurs. Since **EOF** is a valid integer value, **feof** and **ferror** should be used to check the success of **getw**. The **getw** subroutine assumes no special alignment in the file.

Because of possible differences in word length and byte ordering from one machine architecture to another, files written using **putw** are machine-dependent and may not be readable using **getw** on a different type of processor.

Return Value

These subroutines and macros return the integer constant **EOF** at end-of-file or upon an error.

Related Information

In this book: “**feof**, **ferror**, **clearerr**, **fileno**” on page 3-154, “**fopen**, **freopen**, **fdopen**” on page 3-157, “**fread**, **fwrite**” on page 3-177, “**gets**, **fgets**” on page 3-205, “**putc**, **putchar**, **fputc**, **putw**” on page 3-266, “**scanf**, **fscanf**, **sscanf**” on page 3-281, and “**stdio**” on page 3-297.

getcwd

Purpose

Gets the path name of the current directory.

Library

Standard C Library (**libc.a**)

Syntax

```
char *getcwd (buf, size)  
char *buf;  
int size;
```

Description

The **getcwd** subroutine returns a pointer to a string containing the path name of the current directory. The value of the *size* parameter must be at least two greater than the length of the path name to be returned.

If the *buf* parameter is a **NULL** pointer, the **getcwd** subroutine will, using the **malloc** subroutine, obtain the number of bytes of free space as specified by the *size* parameter. In this case, the pointer returned by the **getcwd** subroutine can be used as the parameter in a subsequent call to **free**.

The function is implemented by using **popen** to pipe the output of the **pwd** command into the specified string space.

If the **getcwd** subroutine fails, **NULL** is returned and **errno** is set to indicate the error. The **getcwd** subroutine fails if the *size* parameter is not large enough or if an error occurs in a lower-level function.

Related Information

In this book: “malloc, free, realloc, calloc” on page 3-224 and “popen, pclose” on page 3-260.

The **pwd** command in *AIX Operating System Commands Reference*.

getenv

Purpose

Returns the value of an environment variable.

Library

Standard C Library (**libc.a**)

Syntax

```
char *getenv (name)
char *name;
```

Description

The **getenv** subroutine searches the environment list for a string of the form *name = value*. Environment variables are sometimes called shell variables since they are frequently set with shell commands.

Return Value

The **getenv** subroutine returns a pointer to the *value* in the current environment if such a string is present. If such a string is not present, a **NULL** pointer is returned.

Related Information

In this book: “environment” on page 5-44.

The **sh** command in *AIX Operating System Commands Reference*.

getgrent, getgrgid, getgrnam, setgrent, endgrent

getgrent, getgrgid, getgrnam, setgrent, endgrent

Purpose

Accesses group file entries.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <grp.h>
```

```
struct group *getgrent ( )
```

```
struct group *getgrgid (gid)  
int gid;
```

```
struct group *getgrnam (name)  
char *name;
```

```
void setgrent ( )
```

```
void endgrent ( )
```

Description

The **getgrent**, **getgrgid**, and **getgrnam** subroutines return a pointer to a structure containing the broken-out fields of a line in the **/etc/group** file. The **group** structure is defined in the **grp.h** header file, and it contains the following members:

```
char *gr_name;    /* The name of the group */  
char *gr_passwd;  /* The encrypted group password */  
int gr_gid;       /* The numerical group ID */  
char **gr_mem;    /* Array of pointers to member names */
```

The **getgrent** subroutine, when first called, returns a pointer to the first group structure in the file. On the next call, it returns a pointer to the next group structure in the file. You can call **getgrent** repeatedly to search the entire file.

The **getgrgid** subroutine searches from the beginning of the file until it finds a numerical group ID matching the *gid* parameter. The subroutine then returns a pointer to the structure in which it was found.

getgrent, getgrgid, getgrnam, setgrent, endgrent

The **getgrnam** subroutine searches from the beginning of the file until it finds a group name matching the *name* parameter. The subroutine then returns a pointer to the structure in which it was found.

If an end-of-file condition or an error is encountered on reading, these functions return a **NULL** pointer.

The **setgrent** subroutine rewinds the group file to allow repeated searches.

The **endgrent** subroutine closes the group file when processing is complete.

Warning: All information is contained in a static area, so it must be copied if it is to be saved.

File

`/etc/group`

Related Information

In this book: “getlogin” on page 3-196, “getpwent, getpwuid, getpwnam, setpwent, endpwent” on page 3-203, and “group” on page 4-82.

getlogin

getlogin

Purpose

Gets the user's login name.

Library

Standard C Library (**libc.a**)

Syntax

char *getlogin ()

Description

The **getlogin** subroutine returns a pointer to the login name as found in **/etc/utmp**. Use the **getlogin** subroutine in conjunction with the **getpwnam** subroutine to locate the correct password file entry when the same user ID is shared by several login names.

If the **getlogin** subroutine is called within a process that is not attached to a terminal, it returns a **NULL** pointer. The correct procedure for determining the login name is to call **cuserid**, or to call **getlogin** and if it fails, then to call **getpwuid**.

If the login name is not found, **getlogin** returns a **NULL** pointer.

Warning: The **getlogin** subroutine returns a pointer to a static area that is overwritten by successive calls.

File

/etc/utmp

Related Information

In this book: “cuserid” on page 3-62, “getgrent, getgrgid, getgrnam, setgrent, endgrent” on page 3-194, “getpwent, getpwuid, getpwnam, setpwent, endpwent” on page 3-203, and “utmp” on page 4-163.

getopt

getopt

Purpose

Gets flag letters from the argument vector.

Library

Standard C Library (**libc.a**)

Syntax

```
int getopt (argc, argv, optstring)  
int argc;  
char **argv;  
char *optstring;
```

```
extern char *optarg;  
extern int optind;
```

Description

The **getopt** subroutine returns the next flag letter in the *argv* parameter list that matches a letter in the *optstring* parameter. The **getopt** subroutine is an aid to help programs interpret shell command-line flags that are passed to them.

The *optstring* parameter is a string of recognized flag letters. If a letter is followed by a colon, the flag is expected to take a parameter that may or may not be separated from it by white space. The **optarg** external variable is set to point to the start of the flag's parameter on return from the **getopt** subroutine.

The **getopt** subroutine places the *argv* index of the next argument to be processed in **optind**. **optind** is externally initialized to 1 so that *argv*[0] is not processed.

When all flags have been processed (that is, up to the first nonflag argument), the **getopt** subroutine returns **EOF**. The special flag -- (dash dash) can be used to delimit the end of the flags; **EOF** is returned, and -- is skipped.

The **getopt** subroutine prints an error message on **stderr** and returns (int) '?' (question mark) when it encounters a flag letter that is not included in the *optstring* parameter.

Examples

The following code fragment processes the flags for a command that can take the mutually exclusive flags **a** and **b**, and the flags **f** and **o**, both of which require parameters.

```
#include <unistd.h>      /* Needed for access system call constants */

main (argc, argv)
int argc;
char **argv;
{
    int c;
    extern int optind;
    extern char *optarg;
    .
    .
    .
    while ((c = getopt(argc, argv, "abf:o:")) != EOF)
    {
        switch (c)
        {
            case 'a':
                if (bflg)
                    errflg++;
                else
                    aflg++;
                break;
            case 'b':
                if (aflg)
                    errflg++;
                else
                    bflg++;
                break;
            case 'f':
                ifile = optarg;
                break;
            case 'o':
                ofile = optarg;
                break;
```

getopt

```
        case '?':
            errflg++;
        } /* case */

        if (errflg)
        {
            fprintf(stderr, "usage: . . . ");
            exit(2);
        }
    } /* while */

    for ( ; optind < argc; optind++)
    {
        if (access(argv[optind], R_OK))
        {
            .
            .
            .
        }
    } /* for */
} /* main */
```

Related Information

The **getopt** command in *AIX Operating System Commands Reference*.

getpass

Purpose

Reads a password.

Library

Standard C Library (**libc.a**)

Syntax

```
char *getpass (prompt)  
char *prompt;
```

Description

The **getpass** subroutine writes the *prompt* string to standard error output, disables echoing, and reads up to a new-line character or EOF from the file **/dev/tty**.

It returns a pointer to a null-terminated string of no more than 8 characters. This return value points to data that is overwritten by successive calls. If the **/dev/tty** file cannot be opened, a **NULL** pointer is returned.

An interrupt terminates input and sends an interrupt signal to the calling program before returning.

File

/dev/tty

Related Information

In this book: “crypt, encrypt” on page 3-39.

getpw

getpw

Purpose

Get name from UID.

Library

Standard C Library (**libc.a**)

Syntax

```
int getpw (uid, buf)
int uid;
char *buf;
```

Description

The **getpw** subroutine is included only for compatibility with prior systems and should not be used unless your program is going to be used with a prior system. See “getpwent, getpwuid, getpwnam, setpwent, endpwent” on page 3-203 and “putpwent” on page 3-268 for subroutines to use instead.

The **getpw** searches the password file for a user ID number that matches the *uid* parameter. When a match is found, **getpw** copies the line of the password file in which the match was found into an array pointed to by the *buf* parameter. The subroutine then returns a value of 0. If a match cannot be found, the subroutine returns a nonzero value.

File

/etc/passwd

Related Information

In this book: “passwd” on page 4-107.

getpwent, getpwuid, getpwnam, setpwent, endpwent

Purpose

Gets a password file entry.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <pwd.h>
```

```
struct passwd *getpwent ( )
```

```
struct passwd *getpwuid (uid)  
int uid;
```

```
struct passwd *getpwnam (name)  
char *name;
```

```
void setpwent ( )
```

```
void endpwent ( )
```

Description

The **getpwent**, **getpwuid**, and **getpwnam** subroutines return a pointer to a structure containing the broken-out fields of a line in the **/etc/passwd** file. The **passwd** structure is defined in the **pwd.h** header file, and it contains the following members:

```
char *pw_name;  
char *pw_passwd;  
int pw_uid;  
int pw_gid;  
char *pw_age;  
char *pw_comment;  
char *pw_etc;  
char *pw_dir;  
char *pw_shell;
```

The **pw_comment** field is unused; the others have meanings described in “passwd” on page 4-107.

getpwent, getpwuid, getpwnam, setpwent, endpwent

The **getpwent** subroutine, when first called, returns a pointer to the first **passwd** structure in the file. On the next call, it returns a pointer to the next **passwd** structure in the file. Successive calls can be used to search the entire file.

The **getpwuid** subroutine searches from the beginning of the file until it finds a numerical user ID matching the *uid* parameter. The subroutine then returns a pointer to the structure in which it was found.

The **getpwnam** subroutine searches from the beginning of the file until it finds a login name matching the *name* parameter. The subroutine then returns a pointer to the structure in which it was found.

If an end-of-file condition or an error is encountered on reading, these functions return a **NULL** pointer.

The **setpwent** subroutine rewinds the password file to allow repeated searches.

The **endpwent** subroutine closes the group file when processing is complete.

Warning: All information is contained in a static area, so it must be copied if it is to be saved.

File

`/etc/passwd`

Related Information

In this book: “getlogin” on page 3-196 and “getgrent, getgrgid, getgrnam, setgrent, endgrent” on page 3-194.

gets, fgets

Purpose

Gets a string from a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
char *gets (s)  
char *s;
```

```
char *fgets (s, n, stream)  
char *s;  
int n;  
FILE *stream;
```

Description

The **gets** subroutine reads characters from the standard input stream, **stdin**, into the array pointed to by the *s* parameter. Data is read until a new-line character is read or an end-of-file condition is encountered. If reading is stopped due to a new-line character, the new-line character is discarded and the string is terminated with a null character.

The **fgets** subroutine reads characters from the data pointed to by the *stream* parameter into the array pointed to by the *s* parameter. Data is read until *n* - 1 characters have been read, until a new-line character is read and transferred to *s*, or until an end-of-file condition is encountered. The string is then terminated with a null character.

Return Value

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a **NULL** pointer is returned. If a read error occurs, a **NULL** pointer is returned. Otherwise, *s* is returned.

Related Information

In this book: “feof, ferror, clearerr, fileno” on page 3-154, “fopen, freopen, fdopen” on page 3-157, “fread, fwrite” on page 3-177, “getc, fgetc, getchar, getw” on page 3-189, “puts, fputs” on page 3-269, “scanf, fscanf, sscanf” on page 3-281, and “stdio” on page 3-297.

getuinfo

Purpose

Finds the value associated with a user information name.

Library

Standard C Library (**libc.a**)

Syntax

```
char *getuinfo (name)
char *name;
```

Description

The **getuinfo** subroutine searches a user information buffer for a string of the form *name=value* and returns a pointer to the *value* substring if *name* is found. **NULL** is returned if *name* is not found.

The user information buffer searched is pointed to by the global variable:

```
extern char *INuibp;
```

This variable is initialized to **NULL**.

If the variable **INuibp** is **NULL** when the **getuinfo** subroutine is called, the **usrinfo** system call is executed to read user information from the kernel into a local buffer. The address of the buffer is then put into the external variable **INuibp**. The **usrinfo** system call is automatically called the first time the **getuinfo** subroutine is called if the **INuibp** variable has not been set.

Related Information

In this book: “usrinfo” on page 2-170.

getutent

getutent

Purpose

Accesses **utmp** file entries.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <utmp.h>
```

```
struct utmp *getutent ( )
```

```
struct utmp *getutid (id)
struct utmp *id;
```

```
struct utmp *getutline (line)
struct utmp *line;
```

```
void pututline (utmp)
struct utmp *utmp;
```

```
void setutent ( )
```

```
void endutent ( )
```

```
void utmpname (file)
char *file;
```

Description

The **getutent**, **getutid**, and **getutline** subroutines each return a pointer to a structure of the following type:

```
#define ut_name    ut_user
#define ut_id      ut_line
```

```
struct utmp
{
```

```
    char    ut_user[8];    /* User name                */
    char    ut_line[12];   /* Device name (console, lxxx) */
    short   ut_pid;        /* Process ID                */
    short   ut_type;       /* Type of entry              */
    struct  exit_status
```

```
    {
        short  e_termination; /* Process termination status */
        short  e_exit;        /* Process exit status */
    } ut_exit;                /* The exit status of a DEAD_PROCESS */
time_t      ut_time;        /* Time entry was made */
};
```

The **getutent** subroutine reads the next entry from a **utmp**-like file. If the file is not already open, this subroutine opens it. If the end of the file is reached, **getutent** fails.

If you specify type **RUN_LVL**, **BOOT_TIME**, **OLD_TIME**, or **NEW_TIME** in the *id* parameter, the **getutid** subroutine searches forward from the current point in the **utmp** file until an entry with a **ut_type** matching *id->ut_type* is found.

If you specify one of the types **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS** or **DEAD_PROCESS** in the *id* parameter, then the **getutid** subroutine returns a pointer to the first entry whose type is one of these four and whose **ut_id** field matches *id->ut_id*. If the end of the file is reached without a match, the **getutid** subroutine fails.

The **getutline** subroutine searches forward from the current point in the **utmp** file until it finds an entry of the type **LOGIN_PROCESS** or **USER_PROCESS** that also has a **ut_line** string matching the *line->ut_line* parameter string. If the end of the file is reached without a match, the **getutline** subroutine fails.

The **pututline** subroutine writes the supplied **utmp** structure into the **utmp** file. If you have not searched for the proper place in the file using one of the **getut** routines, then the **pututline** subroutine calls **getutid** to search forward for the proper place. It is expected that normally the user of **pututline** searched for the proper entry using one of the **getut** subroutines. If so, **pututline** does not search. If the **pututline** subroutine does not find a matching slot for the entry, it adds a new entry to the end of the file.

The **setutent** subroutine resets the input stream to the beginning of the file. You should do this before each search for a new entry if you want to examine the entire file.

The **endutent** subroutine closes the currently open file.

The **utmpname** subroutine changes the name of the file to be examined from **/etc/utmp** to any other file. The name specified is usually **/usr/adm/wtmp**. If the specified file does not exist, no indication is given. You are not aware of this fact until your first attempt to reference the file. The **utmpname** subroutine does not open the file. It closes the old file, if it is currently open, and saves the new file name.

The most current entry is saved in a static structure. If you desire to make multiple accesses, you must copy or use the structure between each access. The **getutid** and **getutline** subroutines examine the static structure first. If the contents of the static structure match what they are searching for, they do not read the **utmp** file. Therefore, you must fill the static structure with zeroes after each use if you want to use these subroutines to search for multiple occurrences.

getutent

If **pututline** finds that it isn't already at the correct place in the file, then the implicit read it performs does not overwrite the contents of the static structure returned by the **getutent**, **getuid**, or **getutline** routine. This allows you to get an entry with one of these subroutines, modify the structure, and pass the pointer back to **pututline** for writing.

These subroutines use buffered standard I/O for input, but **pututline** uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the **utmp** and **wtmp** files.

Return Value

These subroutines fail and return a **NULL** pointer if a read or write fails due to end-of-file or a permission conflict.

Files

/etc/utmp
/usr/adm/wtmp

Related Information

In this book: “**ttyslot**” on page 3-319, and “**utmp**” on page 4-163.

hsearch, hcreate, hdestroy

Purpose

Manages hash tables.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <search.h>
```

```
ENTRY *hsearch (item, action)  
ENTRY item;  
ACTION action;
```

```
int hcreate (nel)  
unsigned int nel;
```

```
void hdestroy ( )
```

Description

The **hsearch** subroutine is a hash table search routine. It returns a pointer into a hash table that indicates the location of a given entry. The *item* parameter is a structure of the type **ENTRY** as defined in the **search.h** header file. It contains two pointers:

item.key Points to the comparison key.

item.data Points to any other data be associated with that key.

Pointers to types other than **char** should be cast to pointer-to-character. The *action* parameter is a value of the **ACTION** enumeration type that indicates what is to be done with an entry if it cannot be found in the table:

ENTER Enters the item into the table at the appropriate point. If the table is full, a **NULL** pointer is returned.

FIND Does not enter the item into the table, but returns a **NULL** pointer if the item cannot be found.

The **hsearch** subroutine uses open addressing with a multiplicative hash function.

The **hcreate** subroutine allocates sufficient space for the table. You must call **hcreate** before calling **hsearch**. The *nel* parameter is an estimate of the maximum number of entries that the table contains. Under some circumstances, **hcreate** may actually make

hsearch, hcreate, hdestroy

the table larger than specified. Upon successful completion, **hcreate** returns 1. **hcreate** returns 0 if it cannot allocate sufficient space for the table.

The **hdestroy** subroutine deletes the hash table. This allows you to start a new hash table since only one table can be active at a time.

Related Information

In this book: “bsearch” on page 3-11, “lsearch” on page 3-222, “string” on page 3-299, and “tsearch, tdelete, twalk” on page 3-316.

hypot

Purpose

Computes the euclidean distance function.

Library

Math Library (**libm.a**)

Syntax

```
#include <math.h>
```

```
double hypot (x, y)  
double x, y;
```

Description

The **hypot** subroutine takes precautions against overflows while computing the value of:

$$\sqrt{x^2 + y^2}$$

If the correct value *does* overflow, then **hypot** returns **HUGE** and sets **errno** to **ERANGE**.

You can change the error-handling procedures by supplying a **matherr** subroutine. See “**matherr**” on page 3-226 for more information.

Related Information

In this book: “exp, log, log10, pow, sqrt” on page 3-121.

initgroups

initgroups

Purpose

Initializes group access list.

Library

Standard C Library (**libc.a**)

Syntax

```
int initgroups (user, basegid)  
char *user;  
int basegid;
```

Description

The **initgroups** subroutine reads the **/etc/group** file and constructs the group access list for the user whose name is specified by the *user* parameter. The *basegid* parameter is usually the group number from the **/etc/password** file and, it is automatically included in the group list.

Warning: The **initgroups** subroutine uses the **getgrent** subroutines. If the program that invokes **initgroups** uses any of these subroutines, then calling **initgroups** overwrites the static group structure.

Return Value

Upon successful completion, the **initgroups** subroutine returns a value of 0. If the effective user ID of the calling process is not superuser, then **initgroups** returns a value of 1.

File

`/etc/group`

Related Information

In this book: “getgrent, getgrgid, getgrnam, setgrent, endgrent” on page 3-194, “getgroups” on page 2-50, and “setgroups” on page 2-120.

The **adduser** command in *AIX Operating System Commands Reference*.

l3tol, ltol3

l3tol, ltol3

Purpose

Converts between 3-byte integers and long integers.

Library

Standard C Library (**libc.a**)

Syntax

```
void l3tol (lp, cp, n)  
long *lp;  
char *cp;  
int n;
```

```
void ltol3 (cp, lp, n)  
char *cp;  
long *lp;  
int n;
```

Description

The **l3tol** subroutine converts a list of *n* 3-byte integers packed into a character string pointed to by the *cp* parameter into a list of long integers pointed to by the *lp* parameter.

The **ltol3** subroutine performs the reverse conversion, from long integers (*lp*) to 3-byte integers (*cp*).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

Warning: The numerical values of the long integers are machine-dependent because of possible differences in byte ordering.

Related Information

In this book: “fs” on page 4-72.

libPW

Purpose

Provides subroutines for compatibility with existing programs.

Library

Programmers Workbench Library (**libPW.a**)

Description

The **libpw** subroutines are provided only for compatibility with existing programs. Their use in new programs is not recommended.

alloca (*nbytes*)

Allocates *nbytes* of automatic memory.

any (*c*, *s*)

Determines whether the string *s* contains the character *c*.

anystr (*s1*, *s2*)

Determines the offset in string *s1* of the first character that also occurs in string *s2*.

balbrk (*s*, *open*, *close*, *end*)

Determines the offset in string *s* of the first character in the string *end* that occurs outside of a balanced string as defined by *open* and *close*.

cat (*dest*, *source1*, . . . , 0)

Concatenates the *source* strings and copy them to *dest*.

clean-up ()

Defaults the cleanup routine.

curdir (*s*)

Puts the full path name of the current directory in the string *s*.

dname (*p*)

Determines which directory contains the file *p*.

fatal (*msg*)

General purpose error handler.

fdfopen (*fd*, *mode*)

Same as the **stdio fdopen** subroutine.

- giveup** (*dump*)
Forces a core dump.
- imatch** (*pref, s*)
Determines if the string *pref* is an initial substring of the string *s*.
- index** (*s1, s2*)
Determines the offset of the first occurrence in string *s1* of string *s2*.
- lockit** (*lockfile, count, pid*)
Creates a lock file.
- move** (*s1, s2, n*)
Copies the first *n* characters of string *s1* to string *s2*.
- patoi** (*s*)
Converts string *s* to **int**.
- patol** (*s*)
Converts string *s* to **long**.
- rename** (*oldname, newname*)
Renames the file *oldname* to *newname*.
- repeat** (*dest, s, n*)
Sets *dest* to the string *s* repeated *n* times.
- repl** (*s, old, new*)
Replaces each occurrence of the character *old* in string *s* with the character *new*.
- satoi** (*s, ip*)
Converts string *s* to int and save it in **ip*.
- setsig** (*)*
Causes signals to be caught by **setsig1**.
- setsig1** (*sig*)
General purpose signal handling routine.
- sname** (*s*)
Gets a pointer to the simple name of full path name *s*.
- strend** (*s*)
Finds the end of the string *s*.
- substr** (*s, dest, origin, len*)
Places a substring of string *s* in *dest* using the offset *origin* and the length *len*.
- trnslat** (*s, old, new, dest*)
Copies string *s* into *dest* and replace any character in *old* with the corresponding characters in *new*.

- unlockit** (*lockfile, pid*)
Deletes the lock file.
- userdir** (*uid*)
Gets the user's login directory.
- userexit** (*code*)
Defaults user exit routine.
- username** (*uid*)
Gets the user's login name.
- verify** (*s1, s2*)
Determines the offset in string *s1* of the first character that is not also in string *s2*.
- xalloc** (*asize*)
Allocates memory.
- xcreat** (*name, mode*)
Creates a file.
- xfree** (*aptr*)
Frees memory.
- xfreeall** (*)*
Frees all memory.
- xlink** (*f1, f2*)
Links files.
- xmsg** (*file, func*)
Calls the routine **fatal** with an appropriate error message.
- xopen** (*name, mode*)
Opens a file.
- xpipe** (*t*)
Creates a pipe.
- xunlink** (*f*)
Removes a directory entry.
- xwrite** (*fd, buffer, n*)
Writes *n* bytes to the file associated with *fd* from *buffer*.
- zero** (*p, n*)
Zeroes *n* bytes starting at address *p*.
- zeropad** (*s*)
Replaces the initial blanks with the character '0' in string *s*.

Related Information

In this book: “logname” on page 3-221 and “regcmp, regex” on page 3-274.

logname

Purpose

Returns the login name of the user.

Library

Programmers Workbench Library (**libPW.a**)

Syntax

```
char *logname ( )
```

Description

The **logname** subroutine returns a pointer to the null-terminated login name. The **logname** subroutine extracts the **LOGNAME** variable from the user's environment.

Note: The return values point to static data whose content is overwritten by each call. This method of determining a login name is subject to forgery. For better methods, see "cuserid" on page 3-62, "getlogin" on page 3-196, and "getpwent, getpwuid, getpwnam, setpwent, endpwent" on page 3-203.

File

/etc/profile

Related Information

In this book: "profile" on page 4-121 and "environment" on page 5-44.

The **env** and **login** commands in *AIX Operating System Commands Reference*.

lsearch

lsearch

Purpose

Performs a linear search and update.

Library

Standard C Library (**libc.a**)

Syntax

```
char *lsearch ((char *)key, (char *)base, nelp, sizeof (*key), compar)
unsigned int *nelp;
int (*compar) ( );
```

Description

The **lsearch** subroutine is a linear search routine generalized from Donald E. Knuth's *The Art of Computer Programming*, Volume 3, 6.1, Algorithm S.* It returns a pointer into a table indicating where a datum can be found. If the datum does not occur, it is added at the end of the table.

The *key* parameter points to the datum to be sought in the table. The *base* parameter points to the first element in the table. The *nelp* parameter points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. The *compar* parameter is the name of the comparison function that you must supply (**stremp**, for example). It is called with two arguments that point to the elements being compared. The *compar* function must return a value of 0 if the elements are equal and nonzero if they are not equal.

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

* Reading, Massachusetts: Addison-Wesley, 1981.

Warning: Undefined results can occur if there is not enough room in the table to add a new item.

Related Information

In this book: “bsearch” on page 3-11, “hsearch, hcreate, hdestroy” on page 3-211, and “tsearch, tdelete, twalk” on page 3-316.

malloc, free, realloc, calloc

malloc, free, realloc, calloc

Purpose

Provides a memory allocator.

Library

Standard C Library (**libc.a**)

Syntax

```
char *malloc (size)  
unsigned int size;
```

```
void free (ptr)  
char *ptr;
```

```
char *realloc (ptr, size)  
char *ptr;  
unsigned int size;
```

```
char *calloc (nelem, elsize)  
unsigned int nelem, elsize;
```

Description

The **malloc** and **free** subroutines provide a simple general-purpose memory allocation package.

The **malloc** subroutine returns a pointer to a block of memory of at least *size* bytes. The block is aligned so that it can be used for any type of data. Undefined results occur if the space assigned by **malloc** is overrun.

The **malloc** subroutine searches memory for the first contiguous area of free space of at least *size* bytes. The search is performed in a circular pattern from the last block of memory allocated or freed. During the search, **malloc** joins adjacent free blocks of memory. If a large enough contiguous area of free space is not found, then **malloc** issues a **sbrk** system call to get more memory from the system.

The **free** subroutine frees the block of memory pointed to by the *ptr* parameter for further allocation. The block pointed to by the *ptr* parameter must have been previously allocated by the **malloc** subroutine. The **free** subroutine does not change the contents of this block of memory. Undefined results occur if the *ptr* parameter is not a valid pointer.

The **realloc** subroutine changes the size of the block of memory pointed to by the *ptr* parameter to the number of bytes specified by the *size* parameter, and then it returns a pointer to the block. The contents of the block remain unchanged up to the lesser of the old and new sizes. If a large enough block of memory is not available, then **realloc** calls

the **malloc** subroutine to enlarge the memory arena, and then moves the data to the new space.

The **realloc** subroutine also works if the *ptr* parameter points to a block freed since the last call to **malloc**, **realloc**, or **calloc**.

The **calloc** subroutine allocates space for an array with the number of elements specified by the *nelem* parameter. Each element is of the size specified by the *elsize* parameter. The space is initialized to 0's.

Each of the allocation subroutines returns a pointer to space suitably aligned for storage of any type of object. Cast the pointer to the type *pointer-to-element* before using it.

The **malloc**, **realloc**, and **calloc** subroutines return a **NULL** pointer if there is no available memory or if the memory arena has been corrupted by storing outside the bounds of a block. When this happens, the block pointed to by the *ptr* parameter could be destroyed.

matherr

matherr

Purpose

Performs an action when a math subroutine encounters an error.

Library

Math Library (**libm.a**)

Syntax

```
#include <math.h>

int matherr (excp)
struct exception *excp;
```

Description

The math library subroutines call **matherr** whenever an error is detected. You can override the default error-handling actions by using a subroutine of your own in place of the **matherr** subroutine supplied in the math library. To do this, include in your program a subroutine named **matherr** that takes one parameter: a pointer to an **exception** structure. The **exception** structure is defined in the **math.h** header file and it contains the following members:

```
int      type;
char     *name;
double   arg1, arg2, retval;
```

The structure member named **type** describes the type of error that occurred. Its value is one of the following constants:

DOMAIN	Domain error
SING	Singularity
OVERFLOW	Overflow
UNDERFLOW	Underflow
TLOSS	Total loss of significance
PLOSS	Partial loss of significance

The **name** member points to a string containing the name of the subroutine that encountered the error. The members **arg1** and **arg2** contain the parameters that were passed to the subroutine. The **retval** member is the value that the math subroutine returns.

All of the math subroutines call **matherr** in ways similar to this:

```
/*
** Set up the exception structure
*/
exc.type = DOMAIN;    /* Type of error          */
exc.name = "pow";      /* Name of subroutine    */
exc.arg1 = x;          /* Arguments to pow(x,y) */
exc.arg2 = y;

if (matherr(&exc) == 0)
{
    /*
    ** matherr returned 0, so perform the
    ** default error-handling procedures
    */
    fprintf(stderr, "pow: DOMAIN error\n");
    exc.retval = 0;
    errno = EDOM;
}

return (exc.retval);
```

Studying this sample shows that the return value from the **matherr** subroutine controls whether or not the math subroutine performs its default error-handling procedures. If **matherr** returns 0, then the default procedures are performed. Note in particular that if you want to specify the value to be returned by the math subroutine, then your **matherr** subroutine must set *exc->retval* and return a nonzero value.

If you do not supply your own **matherr** subroutine, then the **matherr** subroutine supplied in the math library is linked into your program. This subroutine does nothing except return the value 0. Because it returns 0, the calling math subroutine then performs its default error-handling procedures. The default procedures are mentioned in the discussion of each math subroutine.

matherr

The following table shows the default error-handling procedures for the corresponding math library subroutines:

	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
acos	M,0,D ¹	—	—	—	—	—
asin	M,0,D ²	—	—	—	—	—
bessel	M,-H,D ³	—	±H,R ⁴	0,R	—	*,R
cos	—	—	—	—	M,0,R	—
cosh	—	—	H,R	—	—	—
exp	—	—	H,R	0,R	—	—
gamma	M,H,D ⁵	—	H,R	—	—	—
hypot	—	—	H,R	—	—	—
log	M,-H,D ⁶	M,-H,D ⁷	—	—	—	—
pow	M,0,D ⁸	—	H,R	0,R	—	—
sin	—	—	—	—	M,0,R	—
sinh	—	—	H,R	—	—	—
sqrt	M,0,D ⁹	—	—	—	—	—
tan	—	—	±H,R ¹⁰	—	M,0,R	*,R

Figure 3-2. Default Error-Handling Procedures

The following abbreviations are used in the table:

- * As much as possible of the value is returned.
- 0 Zero is returned.
- H **HUGE** is returned.
- H **-HUGE** is returned.
- ±H **HUGE** or **-HUGE** is returned.
- M A message is written to **stdout**.
- D **errno** is set to **EDOM**.
- R **errno** is set to **ERANGE**.

Notes:

- ¹ Caused by passing **acos** a value larger than 1.0.
- ² Caused by passing **asin** a value larger than 1.0.
- ³ Caused by passing **y0**, **y1**, or **yn** a nonpositive value.

- ⁴ **-HUGE** is returned if **y0** is passed a very small positive value. Otherwise, **HUGE** is returned.
- ⁵ Caused by passing **gamma** a nonpositive integer.
- ⁶ Caused by passing **log** a negative value.
- ⁷ Caused by trying to calculate **log(0)**.
- ⁸ Caused by trying to raise a negative number to a noninteger power, or by trying to calculate **pow(0, 0)**.
- ⁹ Caused by passing **sqrt** a negative value.
- ¹⁰ The return value has the same sign as the parameter passed to **tan**.

Examples

The following subroutine suggests the kinds of actions that a user-supplied **matherr** subroutine might perform. It is *not* the **matherr** subroutine that is provided in the math library. The supplied **matherr** subroutine merely returns 0.

```
int matherr(x)
register struct exception *x;
{
    switch (x->type)
    {
        case DOMAIN:
        case SING:
            /* Display message and abort */
            fprintf(stderr, "domain error in %s\n", x->name);
            abort();

        case OVERFLOW:
            if (strcmp("exp", x->name) == 0)
            {
                /* If exp, display message & return the argument */
                fprintf(stderr, "exp of %f\n", x->arg1);
                x->retval = x->arg1;
            }
            else
```

matherr

```
        if (strcmp("sinh", x->name) == 0)
        {
            /* If sinh, set errno, return 0 */
            errno = ERANGE;
            x->retval = 0;
        }
        else
            /* Otherwise, return HUGE */
            x->retval = HUGE;
        break;

case UNDERFLOW:
    return (0); /* Perform the default procedures */

case TLOSS:
case PLOSS:
    /* Display message and return 0 */
    fprintf(stderr, "loss of significance in %s\n",
            x->name);
    x->retval = 0;
    break;
} /* switch */

return (1); /* Do NOT perform the default procedures */
} /* matherr */
```

Related Information

In this book: “math.h” on page 5-52.

mdverify

Purpose

Controls write-verify operation for a minidisk.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <mdverify.h>
```

```
int mdverify (md, req)  
char *md;  
char req;
```

Description

The **mdverify** subroutine turns write-verify operation on or off for a given minidisk. When write-verify operation is turned on, the system checks each write operation to the minidisk. After data is written, the system reads it and compares it with the data in the write buffer. If an uncorrectable error is detected, then an error code is passed back from the write operation.

The *md* parameter is a string that specifies the name of the minidisk as it appears in the */etc/system* file (for example, "hd1"). The *req* parameter is one of the following values:

- 'v' Turns write-verify operation on.
- 'o' Turns write-verify operation off.
- 'q' Queries the current write-verify status.

mdverify

Return Value

Upon successful completion of a 'v' or 'o' request, the **mdverify** subroutine returns a value of **MDV_succ**. A successful 'q' request returns one of the following values:

MDV_wvon Write-verify operation is currently turned on.
MDV_wvoff Write-verify operation is currently turned off.

If the **mdverify** subroutine fails, then it returns one of the following values:

-1 The error is indicated by the value of **errno**.
MDV_open The **/etc/system** file could not be opened.
MDV_nosd The *md* parameter does not specify a valid minidisk name.
MDV_iarg The *req* parameter is not valid.
MDV_csf The IODN could not be found, indicating that the **/etc/system** file has probably been damaged.
MDV_iiodn The minidisk manager detected an invalid minidisk IODN.
MDV_dioe The minidisk manager encountered a disk I/O error.
MDV_iop The minidisk manager encountered an invalid operation mode.

File

/etc/system

Related Information

In this book: “system” on page 4-133 and the **VQUERY** and **VCNTRL ioctl** operations described in “hd” on page 6-16.

The **verify** command in *AIX Operating System Commands Reference*.

memccpy, memchr, memcmp, memcpy, memset

Purpose

Performs memory operations.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <memory.h>
```

```
char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;
```

```
char *memchr (s, c, n)
char *s;
int c, n;
```

```
int memcmp (s1, s2, n)
char *s1, *s2;
int n;
```

```
char *memcpy (s1, s2, n)
char *s1, *s2;
int n;
```

```
char *memset (s, c, n)
char *s;
int c, n;
```

Description

The **memory** subroutines operate on memory areas. A memory area is an array of characters bounded by a count, and not terminated by a null character. The **memory** subroutines do not check for the overflow of any receiving memory area. All of the **memory** subroutines are declared in the **memory.h** header file.

The **memccpy** subroutine copies characters from memory area *s2* into memory area *s1*. The **memccpy** subroutine stops after the first character *c* is copied, or after *n* characters have been copied, whichever comes first. **memccpy** returns a pointer to the character after *c* is copied into *s1*, or a **NULL** pointer if *c* is not found in the first *n* characters of *s2*.

The **memchr** subroutine returns a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s*, or a **NULL** pointer if *c* does not occur.

The **memcmp** subroutine lexicographically compares the first *n* characters in memory area *s1* to the first *n* characters in memory area *s2*. **memcmp** uses native character

memccpy, memchr, memcmp, memcpy, memset

comparison, which may be signed on some machines. The **memcmp** subroutine returns the following values:

Less than 0	If <i>s1</i> is less than <i>s2</i>
Equal to 0	If <i>s1</i> is equal to <i>s2</i>
Greater than 0	If <i>s1</i> is greater than <i>s2</i> .

The **memcpy** subroutine copies *n* characters from memory area *s2* to area *s1*. It returns *s1*.

The **memset** subroutine sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

Warning: Character movement is performed differently in different implementations of these subroutines; thus overlapping moves may yield unexpected results.

Related Information

In this book: “string” on page 3-299 and “swab” on page 3-304.

mktemp

Purpose

Constructs a unique file name.

Library

Standard C Library (**libc.a**)

Syntax

```
char *mktemp (template)
char *template;
```

Description

The **mktemp** subroutine replaces the contents of the string pointed to by the *template* parameter with a unique file name.

The string in the *template* parameter must be a file name with six trailing Xs. The **mktemp** subroutine replaces the Xs with a letter and the current process-ID. The letter is chosen to avoid duplicating the name of an existing file.

Upon successful completion, the **mktemp** subroutine returns the address of the string pointed to by the *template* parameter.

If the string pointed to by the *template* parameter contains no Xs, or if **mktemp** is unable to construct a unique file name from a letter and the current process ID, then the first character of the *template* string is replaced with a null character.

Related Information

In this book: “getpid, getpgrp, getppid” on page 2-52, “tmpfile” on page 3-307, and “tmpnam, tempnam” on page 3-308.

monitor

monitor

Purpose

Starts and stops execution profiling.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mon.h>
```

```
void monitor (lowpc, highpc, shortbuff, bufsize, nfunc)
```

— or —

```
void monitor (0, 0, profbuff, -1, nfunc)
```

```
int (*lowpc) ( ), (*highpc) ( );  
short *shortbuff;  
struct prof *profbuff;  
int bufsize, nfunc;
```

Description

The **monitor** subroutine records a histogram of periodically sampled values of the program counter and counts the number of times certain subroutines are called. The **monitor** subroutine is an interface to the **profil** system call.

Executable programs created with **cc -p** automatically include calls to the **monitor** subroutine. You do not need to call the **monitor** subroutine unless you want fine control over profiling.

If the *bufsize* parameter has any value other than -1, then the parameters to **monitor** are interpreted as shown in the first syntax definition. The *lowpc* parameter specifies the lowest address to be sampled, and the highest address to be sampled is the address just below *highpc*. The *lowpc* parameter cannot be 0 when using the **monitor** subroutine to begin profiling. If **monitor** is called with a *lowpc* value of 0, then monitoring is stopped and the results are written to a file named **mon.out**.

The *shortbuff* parameter points to a user-supplied array of short integers. The number of **shorts** in *shortbuff* is specified by the *bufsize* parameter.

The *nfunc* parameter specifies the maximum number of subroutines whose calls are to be counted. Only calls to functions compiled with the **-p** flag of the **cc** command are recorded.

For the results to be significant, especially for programs with small, heavily-used subroutines, specify a buffer that is no more than a few times smaller than the range of locations sampled.

If *bufsize* has the value -1, then the parameters to **monitor** are interpreted as shown in the second syntax definition. In this case, the arguments *lowpc* and *highpc* are ignored, *nfunc* retains the same meaning as described above, and *profbuff* points to an array of **prof** structures. The **prof** structure is defined in the **mon.h** header file, and it contains the following members:

```
daddr_t  p_low;
daddr_t  p_high;
short_t  *p_buff;
int_t     p_bufsize;
int_t     p_scale;
```

The **monitor** subroutine ignores the value given in **p-scale** and computes a value for it. The **p-high** members in successive structures must be in ascending sequence. The array of structures is terminated with a structure containing a **p-high** member set to zero.

Use the **prof** command to examine the results after executing your program.

Examples

1. To profile the entire program except for floating-point operations:

```
extern etext;

...
monitor ((int (*)( )) 0x10000000, etext, buf, bufsize, nfunc);
```

The identifier **etext** is the address immediately following the program text. (See “end, etext, edata” on page 3-116 for more information about **etext**.)

2. To profile the entire program, including floating-point operations:

```
extern etext;

...
monitor ((int (*)( )) 0x800, etext, buf, bufsize, nfunc);
```

Note that this samples many more instructions, which decreases the resolution of the histogram.

3. To profile an entire program that includes floating-point operations and a shared library:

monitor

```
extern etext;
struct prof buf[4];

buf[0].p_low  = 0x800          /* floating-point text */
buf[0].p_high = 0x1D108
buf[1].p_low  = 0x10000000     /* program text      */
buf[1].p_high = etext
buf[2].p_low  = 0x40000000     /* shared library text */
buf[2].p_high = 0x40030D40
buf[3].p_low  = 0             /* end of array       */
buf[3].p_high = 0
```

```
monitor(0, 0, buf, -1, nfunc);
```

The addresses shown for the shared library text may differ from the ones appropriate for a program you write.

The end of the floating-point text may be different on your system. To determine the correct value to use for `buf[0].p_high` in this example, run the following command:

```
nm -x /unix | grep -fpend
```

This command displays the value of the symbol **-fpend**, which marks the end of the floating-point code in the kernel.

4. To stop execution monitoring and write the results to the file **mon.out**:

```
monitor ((int(*)()) 0);
```

File

mon.out

Related Information

In this book: “profil” on page 2-97 and “end, etext, edata” on page 3-116.

The **cc** and **prof** commands in *AIX Operating System Commands Reference*.

msghelp

Purpose

Issues help text.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <msg00.h>

int msghelp (flags, compid, index [, fildes])
unsigned int flags;
char *compid;
int index, fildes;
```

Description

The **msghelp** subroutine retrieves a predefined help description from a message/insert/help file and then constructs and outputs the help text.

The *flags* parameter allows default help attributes to be overridden. All flag bits for attributes you do not want to override must be off. If no attributes are overridden, the help is written to **stderr**. Attribute override flag bits that can be set are:

MSGFLFIL Writes the help text to the file specified by the *fildes* parameter. If this flag is not set, then the help text is written to **stderr**.

There is no specific flag bit defined for suppressing output of the help ID. If you want to suppress the help ID, do not specify the *displayed component ID* and *displayed help ID* fields of the help description in the message/insert/help file. If the help ID is suppressed, then the help text is aligned *fildes* parameter causes the help text to be aligned at the left margin instead of to the right of the help ID. This allows a full 79-character width, but does not provide component and help IDs for referencing an explanation of the help in a reference manual.

msghelp

The *compid* parameter points to a six-character string that identifies the message/insert/help file where the help control information resides. The *compid* parameter is either:

xxxxcc For a component file, where, by convention:

- xxx* Identifies the software provider or product. IBM reserves the use of the identifiers **COM**, **com**, **SYc**, **sys**, **IBc**, and **ibc**, where *c* is any alphanumeric character.
- ccc* Identifies the particular software component.

common For the common message/insert/help file.

The *index* parameter is an index into the file specified by the *compid* parameter. The *index* parameter is an integer value from 1 to 999 and identifies which help description in the file is to be used.

The *fildes* parameter is an integer file descriptor number indicating the opened file to which the help is to be sent. The *fildes* parameter is used only if the **MSGFLFIL** flag is on.

Return Value

Upon successful completion, a value of 0 is returned. If the **msghelp** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file:

MSG-CPID	The <i>compid</i> parameter is not six characters long. The request is ignored.
MSG-INDX	The <i>index</i> parameter is not in the range of 1 to 999. The request is ignored.
MSG-TABP	The MSGFLTAB flag is on. Since helps cannot reside in a message/insert table, this is not a valid flag for the msghelp subroutine. The request is ignored.
MSG-ALLO	The necessary Message Services work area cannot be allocated. The request is ignored.
MSG-SREG	A segment register is not available for mapping a message/insert/help file. The request is ignored.
MSG-COMP	The message/insert/help file specified by the <i>compid</i> parameter cannot be found. Message Services error message 090-002 is output instead.
MSG-INVL	The file specified by the <i>compid</i> parameter is not a valid message/insert/help file. Message Services error message 090-002 is output instead.

- MSG_MTCH** The file specified by the *compid* parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that was specified in the file to the **puttext** command when the component file was built. Message Services error message 090-002 is output instead.
- MSG_NONE** The correct component files are found, but none contain the message description specified by the *index* parameter. Message Services error message 090-002 is output instead.
- MSG_REFN** The requested help description is found but the description references another help description (in the same file) as the source of the text. The referenced help description does not exist. Message Services error message 090-002 is output instead.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a message/insert/help file fails:

- EACCES** Search permission is denied for a directory in the path prefix of the message/insert/help file.
- ENOTDIR** A component of the path name of the message/insert/help file is not a directory.
- EMFILE** Too many files are open for the process.

One of the following values is returned when an attempt to write to the file specified by the *fildes* parameter fails:

- EBADF** The *fildes* parameter does not specify a valid file descriptor that is open for writing.
- EFBIG** The file specified by the *fildes* parameter exceeds the maximum file size or file size limit for the process.

Related Information

In this book: “msgimed” on page 3-242, “msgqued” on page 3-246, “msgtrtv” on page 3-250, and “message” on page 4-100.

msgimed

msgimed

Purpose

Issues an immediate message.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <msg00.h>

int msgimed (flags, compid, index [, sevcode [, errcode [, fildes]])
unsigned int flags;
char *compid;
int index, sevcode, fildes;
long errcode;
```

Description

The **msgimed** subroutine retrieves a predefined message description from a message/insert table or a message/insert/help file and then constructs the message text and outputs it.

The *flags* parameter allows default message attributes to be overridden. All flag bits for attributes you do not want to override must be off. If no attributes are overridden, a message consisting of a message ID (if defined) and message text is written to **stderr**. Attribute override flag bits that can be set are:

- | | |
|-----------------|--|
| MSGFLTAB | Indicates that the <i>compid</i> parameter is a pointer to a message/insert table instead of a pointer to a six-character component ID identifying a message/insert/help file. |
| MSGFLTIM | Includes with the message the time the message was issued. The time is given in 24-hour format. This flag should always be set if the error is logged. |
| MSGFLSEV | Includes a severity code with the message. The severity code value is specified by the <i>sevcode</i> parameter. |

MSGFLERR Includes an error code with the message. The value of the error code is specified by the *errcode* parameter.

MSGFLFIL Writes the message to the file specified by the *fildes* parameter. If this flag is not set, then the message is written to **stderr**.

There is no specific flag bit defined for suppressing output of the message ID. If you want to suppress the message ID, do not specify the **displayed component ID** and the **displayed message ID** fields of the message description in the message/insert table or the message/insert/help file. Suppression of the message ID for a message output to **stderr** or to the output specified by the *fildes* parameter causes the message to be aligned at the left margin instead of to the right of the message ID. This allows a full 79-character width, but does not provide component and message IDs for referencing an explanation of the message in a reference manual.

The *compid* parameter is either a pointer to a message/insert table or identifies the message/insert/help file. If the **MSGFLTAB** flag is set, then the *compid* parameter is a pointer to a message/insert table where the message description resides. If the **MSGFLTAB** flag is not set, then the *compid* parameter identifies the message/insert/help file where the message description resides. In this case, the *compid* parameter is either:

xxxxcc For a component file, where, by convention:

- xxx* Identifies of the software provider or product. IBM reserves the use of the identifiers **COM**, **com**, **SYc**, **syc**, **IBc**, and **ibc**, where *c* is any alphanumeric character.
- ccc* Identifies the particular software component.

common For the common message/insert/help file.

The *index* parameter is an index into the message/insert table or the message/insert/help file specified by the *compid* parameter. The *index* parameter is an integer value from 1 to 999 and identifies which message description is to be used.

The *sevcode* parameter specifies an integer severity code that is output with the message if the **msgflerr** flag is set. The following severity codes have been defined:

- MSGSVSYT** System termination
- MSGSVAPT** Application termination
- MSGSVOPR** Operator-recoverable error
- MSGSVAPR** Application-recoverable error.

If the **MSGFLSEV** flag is not set, and if the *errcode* or *fildes* parameters are specified, then a dummy *sevcode* parameter must be used as a place holder.

The *errcode* parameter is a long integer value that represents an error code with six decimal digits. The error code is output with the message only if the **MSGFLERR** flag is set. The two high-order decimal digits contain the origin code; the four low-order digits contain an application-defined error return code. The origin code is one of the following values:

MSGORIND	Indeterminate origin.
MSGORVDD	Detected in VRM. Indicates a device driver level failure.
MSGORVCK	Detected in VRM. Indicates a check parameter detected failure in VRM.
MSGORVSV	Detected in VRM. Indicates an SVC handler detected failure in VRM.
MSGORUDD	Detected in AIX device driver.
MSGORUKN	Detected in AIX kernel.
MSGORSHL	Detected in shell command.
MSGORRTS	Detected in run-time service or daemon.
MSGORAPP	Detected in application above the application program interface.

If the **MSGFLERR** flag is not set, and if the *fildev* parameter is specified, then a dummy *errcode* parameter must be used as a place holder.

The *fildev* parameter is a file descriptor indicating the opened file to which the message is to be sent. The *fildev* parameter is used only if the **msgflfil** flag is set.

Return Value

Upon successful completion, a value of 0 is returned. If the **msgimed** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file.

MSG_CPID	The <i>compid</i> parameter is not six characters long. The request is ignored.
MSG_INDX	The <i>index</i> parameter is not in the range of 1 to 999. The request is ignored.
MSG_ALLO	The necessary Message Services work area cannot be allocated. The request is ignored.
MSG_SREG	A segment register is not available for mapping a message/insert/help file. The request is ignored.
MSG_BADP	The message/insert table pointer provided does not point to a message/insert table. The request is ignored.
MSG_TABI	The message/insert table that is provided does not contain the requested message. The request is ignored.
MSG_COMP	The message/insert/help file specified by the <i>compid</i> parameter cannot be found. Message Services error message 090-001 is output instead.
MSG_INVL	The file specified by the <i>compid</i> parameter is not a valid message/insert/help file. Message Services error message 090-001 is output instead.
MSG_MTCH	The file specified by the <i>compid</i> parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that is specified

in the file to the **puttext** command when the component file was built. Message Services error message 090-001 is output instead.

MSG_NONE The correct component files are found, but none contain the message description specified by the *index* parameter. Message Services error message 090-001 is output instead.

MSG_REFN The requested message description is found but the description references another message description (in the same file) as the source of the text. The referenced message description does not exist. Message Services error message 090-001 is output instead.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a message/insert/help file fails:

-EACCES Search permission is denied for a directory in the path prefix. The request is ignored.

-ENOTDIR A component of the path prefix is not a directory. The request is ignored.

-EMFILE Too many files are open for the process. The request is ignored.

One of the following values is returned when an attempt to write to the file specified by the *fildes* parameter fails:

-EBADF Not a valid file descriptor open for writing.

-EFBIG The file exceeds the process's file size limit or the maximum file size.

Related Information

In this book: “msghelp” on page 3-239, “msgqued” on page 3-246, “msgstrv” on page 3-250, and “message” on page 4-100.

msgqueued

msgqueued

Purpose

Issues a queued message.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <msg00.h>
```

```
int msgqueued (flags, compid, index [, sevcode [, errcode]])  
unsigned int flags;  
char *compid;  
int index, sevcode;  
long errcode;
```

Description

The **msgqueued** subroutine retrieves a predefined message description from a message/insert table or a message/insert/help file and then constructs the message text and writes it to the queued message file, **/qmsg**.

The queued message file is installed with the AIX Operating System. After installation, the default message queue size of 100 messages can be changed by using an editor to update the four-digit value, located at the beginning of the file, to the desired queue size.

The queued message file contains the last 100 queued messages that have been issued. After the first 100 messages have been queued, each new message added to the queue overlays the oldest message in the queued message file. The message queue is maintained across IPLs.

A maximum of 79 characters of constructed message text is allowed. Fields other than the message text, such as date, time, severity, and error number, are of fixed length and do not count against the 79-character limit. If the constructed message text exceeds 79 characters, the message truncates.

Queued messages are directed to the console operator and are generally system type messages.

The *flags* parameter allows default message attributes to be overridden. All flag bits for attributes you do not want to override must not be set. If no attributes are overridden, then the message consists of the message ID (if defined), the message text, and the date and time the message was issued. Attribute override flag bits that can be set are:

- MSGFLTAB** Indicates that the *compid* parameter is a pointer to a message/insert table instead of a pointer to a six-character component ID identifying a message/insert/help file.
- MSGFLSEV** Includes a severity code with the message. The severity code value is specified by the *sevcode* parameter.
- MSGFLERR** Includes an error code with the message. The error code value must be specified by the *errcode* parameter.

The *compid* parameter is either a pointer to a message/insert table or identifies the message/insert/help file. If the **MSGFLTAB** flag is set, then the *compid* parameter points to a message/insert table where the message description resides. If the **MSGFLTAB** flag is not set, the *compid* parameter identifies the message/insert/help file where the message description resides.

The *index* parameter is an index into the message/insert table or message/insert/help file specified by the *compid* parameter. The *index* parameter is an integer value from 1 to 999 and identifies which message description in the file is to be used.

The *sevcode* parameter specifies an integer severity code that is written with the message if the **MSGFLERR** flag is set. The following severity codes have been defined:

- MSGSVSYT** System termination
- MSGSVAPT** Application termination
- MSGSVOPR** Operator-recoverable error
- MSGSVAPR** Application-recoverable error.

If the **msgflsev** flag is not set, and if the *errcode* parameter is specified, then a dummy *sevcode* parameter must be used as a place holder.

The *errcode* parameter is a long integer value that represents an error code with six decimal digits. The error code is output with the message only if the **MSGFLERR** flag is set. The two high-order decimal digits contain the origin code; the four low-order digits contain an application-defined error return code. The possible values for the origin code are listed in the description. The origin code is one of the following values:

- MSGORIND** Indeterminate origin.
- MSGORVDD** Detected in VRM. Indicates a device driver level failure.
- MSGORVCK** Detected in VRM. Indicates a check parameter detected failure in VRM.
- MSGORVSV** Detected in VRM. Indicates an SVC handler detected failure in VRM.
- MSGORUDD** Detected in AIX device driver.
- MSGORUKN** Detected in AIX kernel.
- MSGORSHL** Detected in shell command.

MSGORRTS	Detected in run-time service or daemon.
MSGORAPP	Detected in application above the application program interface.

Return Value

Upon successful completion, a value of 0 is returned. If the **msgqued** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file:

MSG_CPID	The <i>compid</i> parameter is not six characters long. The request is ignored.
MSG_INDX	The <i>index</i> parameter is not in the range of 1 to 999. The request is ignored.
MSG_ALLO	The necessary Message Services work area cannot be allocated. The request is ignored.
MSG_SREG	A segment register is not available for mapping a message/insert/help file. The request is ignored.
MSG_BADP	The message/insert table pointer provided does not point to a message/insert table. The request is ignored.
MSG_TABI	The message/insert table that is provided does not contain the requested message. The request is ignored.
MSG_COMP	The message/insert/help file specified by the <i>compid</i> parameter cannot be found. Message Services error message 090-001 is output instead.
MSG_INVL	The file specified by the <i>compid</i> parameter is not a valid message/insert/help file. Message Services error message 090-001 is output instead.
MSG_MTCH	The file specified by the <i>compid</i> parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that is specified in the file to the puttext command when the component file was built. Message Services error message 090-001 is output instead.
MSG_NONE	The correct component files are found, but none contain the message description specified by the <i>index</i> parameter. Message Services error message 090-001 is output instead.
MSG_REFN	The requested message description is found but the description references another message description (in the same file) as the source of the text. The referenced message description does not exist. Message Services error message 090-001 is output instead.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a message/insert/help file fails:

- | | |
|-----------------|---|
| -EACCES | Search permission is denied for a directory in the path prefix of the message/insert/help file. |
| -ENOTDIR | A component of the path name of the message/insert/help file is not a directory. |
| -EMFILE | Too many files are open for the process. |

File

/qmsg

Related Information

In this book: “msghelp” on page 3-239, “msgimed” on page 3-242, “msgtrtv” on page 3-250, and “message” on page 4-100.

msgstrv

msgstrv

Purpose

Retrieves a message, insert, or help text.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <msg00.h>
```

```
int msgstrv (flags, compid, index, buf, nbytes)  
unsigned int flags, nbytes;  
char *compid, *buf;  
int index;
```

Description

The **msgstrv** subroutine retrieves a predefined message, insert, or help description from a message/insert/help file or a resident message/insert table, and then constructs the message, insert, or help text as specified and returns the text.

The *flags* parameter allows default attributes to be overridden. All flag bits for attributes you do not want to override must not be set. If no attributes are overridden, insert text is retrieved from a file. Attribute override flag bits that can be set are:

MSGFLTAB Indicates that the *compid* parameter is a pointer to a message/insert table instead of a pointer to a six-character component ID identifying a message/insert/help file. The **MSGFLTAB** flag should not be set if the **MSGFLHLP** flag is set because helps reside only in a message/insert/help file, not in a message/insert table.

MSGFLMSG Retrieves message text instead of insert text.

MSGFLHLP Retrieves help text instead of insert text.

The *compid* parameter is either a pointer to a message/insert table or identifies the message/insert/help file. If the **MSGFLTAB** flag is set, then the *compid* parameter points to a message/insert table where the message or insert description resides. If the **MSGFLTAB** flag is not set, then the *compid* parameter identifies the message/insert/help

file where the message, insert, or help description resides. In this case, the *compid* parameter is either:

xxxxcc For a component file, where, by convention:

- xxx** Identifies of the software provider or product. IBM reserves the use of the identifiers **COM**, **com**, **SYc**, **syc**, **IBc**, and **ibc**, where *c* is any alphanumeric character.
- ccc** Identifies the particular software component.

common For the common message/insert/help file.

The *index* parameter is an index into the message/insert table or message/insert/help file specified by the *compid* parameter. The *index* parameter is an integer value from 1 to 999 and identifies which message, insert, or help description in the file or table is to be used.

The *buf* parameter must be either a pointer to a buffer or a pointer to a structure, depending on the value of the *nbyte* parameter.

- If the *nbyte* parameter is greater than 0, then *buf* parameter points to a buffer where the message, insert, or help text is to be stored.
- If the *nbyte* parameter is equal to 0, then the *buf* parameter points to a **msg__rtrv** structure provided by the requesting program. The **msg__rtrv** is defined as a **typedef** in the **msg05.h** header file.

The *nbyte* parameter is either the size of the buffer pointed to by the *buf* parameter, or 0. The buffer size should include space for a terminating null character. If the *nbyte* parameter is 0, a buffer is allocated by the **msgtrtv** subroutine. The buffer pointer (*msgbufp* in the **msg05.h** header file) returned by the **msgtrtv** subroutine should always be inspected by the requesting program after the returned text has been processed. If the inspection finds other than a **NULL** buffer pointer, the buffer should be freed. This should be done regardless of the value of the return code.

Return Value

Upon successful completion, a positive value is returned. If the **msgtrtv** subroutine fails, it returns a negative value that indicates the reason why the text could not be retrieved.

The value returned upon successful completion is the actual length of the constructed text, not including the terminating null character. The following should be noted concerning the length:

- If the *nbyte* parameter was 0 and help text with a title was retrieved, the length returned is the sum of the title length and the text length, including the null terminators after the title and the text.
- If the *nbyte* parameter was not 0, and the retrieved text is longer than the buffer provided (minus 1 character for the null terminator), the excess text is truncated. The length of the truncated text is included in the length returned. If the return code value

is greater than the length specified by the *nbyte* parameter minus 1, the following considerations should be noted:

- The length of the text returned in the buffer is the length specified by the *nbyte* parameter minus one instead of the return code value.
- The requesting program knows that the retrieved text had to be truncated in order to fit into the buffer provided.

If the **msghelp** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file:

MSG_CPID	The <i>compid</i> parameter is not six characters long. The request is ignored.
MSG_INDX	The <i>index</i> parameter is not in the range of 1 to 999. The request is ignored.
MSG_TABP	Both the msgfltab and msgflhlp flags are on. Since helps cannot reside in a message/insert table, this is not a valid combination of flag bits. The request is ignored.
MSG_ALLO	The necessary Message Services work area cannot be allocated. The request is ignored.
MSG_SREG	A segment register is not available for mapping a message/insert/help file. The request is ignored.
MSG_BADP	The message/insert table pointer provided does not point to a message/insert table. The request is ignored.
MSG_TABI	The message/insert table that is provided does not contain the requested message or insert. The request is ignored.
MSG_COMP	The message/insert/help file specified by the <i>compid</i> parameter cannot be found. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.
MSG_INVL	The file specified by the <i>compid</i> parameter is not a valid message/insert/help file. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.
MSG_MTCH	The file specified by the <i>compid</i> parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that was specified in the file to the puttext command when the component file was built. If a message was specified, then Message Services error message

090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

MSG_NONE The correct component files are found, but none contain the message, insert, or help description specified by the *index* parameter. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

MSG_REFN The requested message, insert, or help description is found but the description references another message, insert, or help description (in the same file) as the source of the text. The referenced message, insert, or help description does not exist. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a message/insert/help file fails:

- EACCES** Search permission is denied for a directory in the path prefix of the message/insert/help file.
- ENOTDIR** A component of the path name of the message/insert/help file is not a directory.
- EMFILE** Too many files are open for the process.

Related Information

In this book: “msghelp” on page 3-239, “msgimed” on page 3-242, “msgqued” on page 3-246, and “message” on page 4-100.

nlist

nlist

Purpose

Gets entries from a name list.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <nlist.h>

int nlist (filename, nl)
char *filename;
struct nlist *nl;
```

Description

The **nlist** subroutine allows a program to examine the name list in the executable file named by the *filename* parameter. It selectively extracts a list of values and places them in the array of **nlist** structures pointed to by the *nl* parameter.

The name list specified by the *nl* parameter consists of an array of structures containing names of variables, types, and values. The list is terminated with an element that has a null string in the **name** structure member. Each variable name is looked up in the name list of the file. If the name is found, the **type** and **value** of the name are inserted in the next two fields. The **type** field is set to 0 unless the file was compiled with the **-g** option. If the name is not found, both the **type** and **value** entries are set to 0.

All entries are set to 0 if the specified file cannot be read or if it does not contain a valid name list.

You can use the **nlist** subroutine to examine the system name list kept in the **/unix** file. By examining this list, you can ensure that your programs obtain current system addresses.

The **nlist.h** header file is automatically included by **a.out.h** for compatibility. However, do not include **a.out.h** if you only need the information necessary to use the **nlist** subroutine. If you do include **a.out.h**, follow the **#include** statement with the line:

```
#undef n_name
```


Return Value

Upon successful completion, a value of 0 is returned. If the **nlist** subroutine fails, a value of -1 is returned.

Related Information

In this book: “a.out” on page 4-5.

The **cc** command in *AIX Operating System Commands Reference*.

perror

perror

Purpose

Writes a message explaining a system call error.

Library

Standard C Library (**libc.a**)

Syntax

```
void perror (s)
char *s;
```

```
extern int errno;
extern char *sys_errlist[ ];
extern int sys_nerr;
```

Description

The **perror** subroutine writes a message on the standard error output that describes the last error encountered by a system call or library subroutine. The error message includes the parameter string *s* followed by a : (colon), a blank, the message, and a new-line character. To be of the most use, the parameter string *s* should include the name of the program that caused the error. The error number is taken from the external variable **errno**, which is set when an error occurs, but is not cleared when a successful call is made. See Appendix A, "Error Codes" on page A-1 for a discussion of **errno** values and their meanings.

To simplify various message formats, the array of message strings **sys_errlist** is provided. Use **errno** as an index into this table to get the message string without the new-line character. The largest message number provided in the table is **sys_nerr**. Be sure to check **sys_nerr** because new error codes may be added to the system before they are added to the table.

Related Information

In this book: “printf, fprintf, sprintf” on page 3-262.

plot

plot

Purpose

Performs graphic output.

Library

Graphics Libraries (**libplot.a**, **libprint.a**, **lib300.a**, and others)

Syntax

void openpl ()

void erase ()

void label (s)
char *s;

void line (x1, y1, x2, y2)
int x1, y1, x2, y2;

void circle (x, y, r)
int x, y, r;

void arc (x, y, x0, y0, x1, y1)
int x, y, x0, y0, x1, y1;

void move (x, y)
int x, y;

void cont (x, y)
int x, y;

void point (x, y)
int x, y;

void linemod (s)
char *s;

void space (x0, y0, x1, y1)
int x0, y0, x1, y1;

void closepl ()

Description

The **plot** subroutine family generates graphic output in a relatively device-independent manner. The **space** subroutine must be used before any of these functions to declare the amount of space necessary. The **openpl** subroutine must be used before any of the others to open the device for writing. The **closepl** subroutine flushes the output.

The **circle** subroutine draws a circle of radius *r* with center at the point (*x*, *y*).

The **arc** subroutine draws an arc of a circle with center at the point (*x*, *y*) between the points (*x0*, *y0*) and (*x1*, *y1*).

String parameters to the **label** and **linemod** subroutines are terminated by null characters and must not contain new-line characters.

See “plot” on page 4-109 for a description of the effect of the remaining functions.

These routines appear in several separate libraries. The routines in the **libplot.a** library generate device-independent output. The **tplot** command interprets this output for a specific device.

The other versions of these routines each generate output for a specific device. You should normally redirect the output of **libprint.a** to the printer. You can save the output of **libprint.a** in a regular file and print it later. See the **tplot** command in *AIX Operating System Commands Reference* for a description of how to do this.

On an IBM Graphics Printer, the horizontal distance between points is not the same as the vertical distance between points. This means that arcs and circles are drawn as ellipses. Similarly, drawing a square (with four calls to the **line** subroutine) produces a rectangle. To adjust for this, call the **space** subroutine with appropriate scaling factors.

Files

/usr/lib/libplot.a	Produces output for tplot filters
/usr/lib/libprint.a	For an IBM PC Graphics Printer
/usr/lib/lib300.a	For DASI 300
/usr/lib/lib300s.a	For DASI 300s
/usr/lib/lib300S.a	For DASI 300S
/usr/lib/lib450.a	For DASI 450
/usr/lib/lib4014.a	For Tektronix 4014

Related Information

In this book: “plot” on page 4-109.

The **graph** and **tplot** commands in *AIX Operating System Commands Reference*.

popen, pclose

popen, pclose

Purpose

Initiates a pipe to or from a process.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
FILE *popen (command, type)  
char *command, *type;
```

```
int pclose (stream)  
FILE *stream;
```

Description

The **popen** subroutine creates a pipe between the calling program and a shell command to be executed.

The *command* parameter points to a null-terminated string containing a shell command line. The *type* parameter points to a null-terminated string containing an I/O mode, either "r" for reading or "w" for writing.

The **popen** subroutine returns a pointer to a **FILE** structure for the stream. If the *type* parameter is "r", you can read from the standard output of the command by reading from the file *stream*. If the *type* parameter is "w", you can write to the standard input of the command by writing to the file *stream*.

Use the **pclose** subroutine to close any stream you have opened with the **popen** subroutine. The **pclose** subroutine waits for the associated process to terminate and then returns the exit status of the command.

Because open files are shared, a type "r" command can be used as an input filter and a type "w" as an output filter.

Warning: If the original processes and the process started with **popen** concurrently read or write a common file, neither should use buffered I/O. If they do, the results are unpredictable.

Some problems with an output filter can be prevented by taking care to flush the buffer with the **fflush** subroutine (see “fclose, fflush” on page 3-152).

The **popen** subroutine returns a **NULL** pointer if files or processes cannot be created, or if the shell cannot be accessed.

The **pclose** subroutine returns -1 if *stream* is not associated with a **popen** command.

Related Information

In this book: “fclose, fflush” on page 3-152, “fopen, freopen, fdopen” on page 3-157, “pipe” on page 2-93, “stdio” on page 3-297, “system” on page 3-305, and “wait” on page 2-176.

printf, fprintf, sprintf

printf, fprintf, sprintf

Purpose

Prints formatted output.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int printf (format [, value, . . . ])
char *format;
```

```
int sprintf (s, format [, value, . . . ])
char *s, *format;
```

```
int fprintf (stream, format [, value, . . . ])
FILE *stream;
char *format;
```

Description

The **printf** subroutine converts, formats, and writes its *value* parameters, under control of the *format* parameter, to the standard output stream **stdout**.

The **fprintf** subroutine converts, formats, and writes its *value* parameters, under control of the *format* parameter, to the output stream specified by its *stream* parameter.

The **sprintf** subroutine converts, formats, and stores its *value* parameters, under control of the *format* parameter, into consecutive bytes starting at the address specified by the *s* parameter. The **sprintf** subroutine places a '\0' (null character) at the end. It is your responsibility to ensure that enough storage space is available to contain the formatted string.

The *format* parameter is a character string that contains two types of objects:

- Plain characters, which are copied to the output stream.
- Conversion specifications, each of which causes zero or more items to be fetched from the *value* parameter list.

If there are not enough items for the *format* in the *value* parameter list, then the results are unpredictable. If more *values* remain after the entire *format* has been processed, they are ignored.

Each conversion specification in the *format* parameter has the following syntax:

1. A % (percent) sign.
2. Zero or more *options*, which modify the meaning of the conversion specification. The *option* characters and their meanings are:
 - The result of the conversion are left-justified within the field.
 - + The result of a signed conversion always begin with a sign (+ or -).
 - blank* If the first character of a signed conversion is not a sign, a blank is prefixed to the result. If both the *blank* and + options appear, then the *blank* option is ignored.
 - # This option specifies that the value is to be converted to an alternate form. For **c**, **d**, **s**, and **u** conversions, the option has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a 0. For **x** and **X** conversions, a nonzero result has 0x or 0X prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result always contain a decimal point, even if no digits follow the decimal point. For **g** and **G** conversions, trailing zeroes are not removed from the result.
3. An optional decimal digit string that specifies the minimum *field width*. If the converted value has fewer characters than the field width, the field is padded on the left to the length specified by the field width. If the left-adjustment option is specified, the field is padded on the right.
4. An optional *precision*. The precision is a . (period) followed by a decimal digit string. If no *precision* is given, it is treated as 0. The *precision* specifies:
 - The minimum number of digits to appear for the **d**, **u**, **o**, **x**, or **X** conversions
 - The number of digits to appear after the decimal point for the **e** and **f** conversions
 - The maximum number of significant digits for the **g** conversion
 - The maximum number of characters to be printed from a string in the **s** conversion.
5. An optional **l** (the letter "ell") specifying that a following **d**, **u**, **o**, **x**, or **X** conversion character applies to a **long** integer *value*.
6. A character that indicates the type of conversion to be applied:
 - d, u, o, x, X** The integer *value* is converted to signed decimal for the **d** conversion character, unsigned decimal for the **u** conversion character, octal for the **o** conversion character, or hexadecimal notation for the **x** and **X** conversion characters. The letters abcdef are used for the **x** conversion and the letters ABCDEF are used for the **X** conversion. The *precision* specifies the minimum number of digits to appear. If the

printf, fprintf, sprintf

value being converted can be represented in fewer digits, it is expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

- f** The **float** or **double** *value* is converted to decimal notation in the format `[-]ddd.ddd`. The number of digits after the decimal point is equal to the *precision* specification. If no *precision* is specified, then 6 digits are output. If the *precision* is 0, then no decimal point appears.
- e, E** The float or double *value* is converted to the form `[-]d.ddde ± dd`. There is one digit before the decimal point and the number of digits after the decimal point is equal to the *precision* specification. If no *precision* is specified, then 6 digits are output. If the *precision* is 0, then no decimal point appears. The **E** conversion character produces a number with **E** instead of **e** before the exponent. The exponent always contains at least two digits.
- g, G** The float or double *value* is printed in the same style as the **f** or **e, E** conversion characters, with the *precision* specifying the number of significant digits. The style used depends on the value converted. Style **e** is used only if the exponent resulting from the conversion is less than -4, or if it is greater than or equal to the *precision*. Trailing zeroes are removed from the result. A decimal point appears only if it is followed by a digit.
- c** The character *value* is printed.
- s** The *value* is taken to be a string (character pointer) and characters from the string are printed until a `'\0'` (null character) is encountered or the number of characters indicated by the *precision* is reached. If no *precision* is specified, it is taken to be infinite so all characters up to the first null character are printed. If the string pointer *value* has a value of 0 or **NULL**, the results are undefined.
- %** Print a **%**. No parameter is converted.

A *field width* or *precision* may be indicated by an ***** (asterisk) instead of a digit string. In this case, an integer *value* parameter supplies the field width or precision. The *value* parameter that is converted for output is not fetched until the conversion letter is reached, so the parameters specifying field width or precision must appear **before** the value (if any) to be converted.

If the result of a conversion is wider than the *field width*, then the field is expanded to contain the converted result. No truncation occurs. However, a small *precision* may cause truncation on the right.

The **e**, **E**, **f**, **g**, and **G** formats represent the special floating-point values as follows:

Quiet NaN	+QNaN or -QNaN
Signalling NaN	+SNaN or -SNaN
$\pm \infty$	+INF or -INF
± 0	+0 or -0

The representation of the plus sign depends on whether the **+** or *blank* formatting option is specified.

Return Value

Upon successful completion, each of these subroutines returns the number of characters in the output string. The value returned by **sprintf** does not include the final `'\0'` character. If an output error occurs, a negative value is returned.

Related Information

In this book: “`ecvt`, `fcvt`, `gcvt`” on page 3-114, “`putc`, `putchar`, `fputc`, `putw`” on page 3-266, “`scanf`, `fscanf`, `sscanf`” on page 3-281, and “`stdio`” on page 3-297.

Examples of using **printf** in *C Language Guide and Reference*.

putc, putchar, fputc, putw

putc, putchar, fputc, putw

Purpose

Writes a character or a word to a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int putc(c, stream)  
char c;  
FILE *stream;
```

```
int putchar(c)  
char c;
```

```
int fputc(c, stream)  
char c;  
FILE *stream;
```

```
int putw(w, stream)  
int w;  
FILE *stream;
```

Description

The **putc** macro writes the character *c* to the output specified by the *stream* parameter. The character is written at the position at which the file pointer is currently pointing, if defined.

The **putchar** macro is the same as the **putc** macro except that **putchar** writes to the standard output.

The **fputc** subroutine works the same as **putc**, but **fputc** is a true subroutine rather than a macro. It runs more slowly than **putc**, but takes less space per invocation.

Because **putc** is implemented as a macro, it treats incorrectly a *stream* parameter with side effects, such as `putc(c, *f++)`. For such cases, use **fputc** instead. Also, use **fputc** whenever you need to pass a pointer to this subroutine as a parameter to another subroutine.

The **putw** subroutine writes the word (**int**) specified by the *w* parameter to the output specified by the *stream* parameter. The word is written at the position at which the file pointer, if defined, is pointing. The size of a word is the size of an integer and varies from

machine to machine. The **putw** subroutine does not assume or cause special alignment of the data in the file.

Because of possible differences in word length and byte ordering, files written using the **putw** subroutine are machine-dependent, and may not be readable using the **getw** subroutine on a different processor.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line-buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen** subroutine causes it to become buffered or line-buffered. Use the **setbuf** subroutine to change the stream's buffering strategy.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as it is written. When an output stream is buffered, many characters are saved and written as a block. When an output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested).

Return Value

Upon successful completion, these functions each return the value written. If these functions fail, they return the constant **EOF**. They fail if the *stream* is not open for writing, or if the output file size cannot be increased. Because **EOF** is a valid integer, you should use the **ferror** subroutine to detect **putw** errors.

Related Information

In this book: “fclose, fflush” on page 3-152, “feof, ferror, clearerr, fileno” on page 3-154, “fopen, freopen, fdopen” on page 3-157, “fread, fwrite” on page 3-177, “getc, fgetc, getchar, getw” on page 3-189, “printf, fprintf, sprintf” on page 3-262, “puts, fputs” on page 3-269, “setbuf” on page 3-285, and “stdio” on page 3-297.

putwent

putpwent

Purpose

Writes a password file entry.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

Description

The **putpwent** subroutine writes a line on the stream specified by the *f* parameter. The stream that is written on matches the format of **/etc/passwd**.

The *p* parameter is a pointer to a **passwd** structure created by the **getpwent**, **getpwuid**, or **getpwnam** subroutines.

Return Value

Upon successful completion, **putpwent** returns a value of 0. If **putpwent** fails, a nonzero value is returned.

Related Information

In this book: “getpwent, getpwuid, getpwnam, setpwent, endpwent” on page 3-203, and “passwd” on page 4-107.

puts, fputs

Purpose

Writes a string to a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int puts (s)  
char *s;
```

```
int fputs (s, stream)  
char *s;  
FILE *stream;
```

Description

The **puts** subroutine writes the null-terminated string pointed to by the *s* parameter, followed by a new-line character, to the standard output stream, **stdout**.

The **fputs** subroutine writes the null-terminated string pointed to by the *s* parameter to the output stream specified by the *stream* parameter. The **fputs** subroutine does not append a new-line character.

Neither subroutine writes the terminating null character.

Return Value

Upon successful completion, the **puts** and **fputs** subroutines return the number of characters written. Both subroutines return **EOF** on an error. This happens if the routines try to write on a file that has not been opened for writing.

Related Information

In this book: “feof, ferror, clearerr, fileno” on page 3-154, “fopen, freopen, fdopen” on page 3-157, “fread, fwrite” on page 3-177, “gets, fgets” on page 3-205, “printf, fprintf, sprintf” on page 3-262, “putc, putchar, fputc, putw” on page 3-266, and “stdio” on page 3-297.

qsort

Purpose

Sorts a table of data in place.

Library

Standard C Library (**libc.a**)

Syntax

```
void qsort ((char *) base, nel, sizeof (*base), compar)
unsigned int nel;
int (*compar) ( );
```

Description

The **qsort** subroutine sorts a table of data in place. It uses the “quicker-sort” algorithm.

The *base* parameter points to the element at the base of the table. The *nel* parameter is the number of elements in the table. The *compar* parameter is the name of the comparison function.

The comparison function must compare its parameters and return a value as follows:

- If the first parameter is less than the second parameter, *compar* must return a value less than 0.
- If the first parameter is equal to the second parameter, *compar* must return 0.
- If the first parameter is greater than the second parameter, *compar* must return a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

Note: The order in the output of two items that compare equal is unpredictable.

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

Related Information

In this book: “bsearch” on page 3-11, “lsearch” on page 3-222, and “string” on page 3-299.

The **sort** command in *AIX Operating System Commands Reference*.

rand, srand

Purpose

Generates pseudo-random numbers.

Library

Standard C Library (**libc.a**)

Syntax

`int rand ()`

`void srand (seed);`
`unsigned int seed;`

Description

The **rand** subroutine generates a random numbers using a multiplicative congruential algorithm. The random-number generator has a period of 2^{32} , and it returns successive pseudo-random numbers in the range from 0 to $2^{15} - 1$.

The **srand** subroutine resets the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

Note: The **rand** subroutine is a very simple random-number generator. Its spectral properties, the mathematical measurement of how “random” the number sequence is, are somewhat weak. See “drand48” on page 3-111 for a more elaborate random-number generator that has better spectral properties.

regcmp, regex

regcmp, regex

Purpose

Compiles and matches regular-expression patterns.

Library

Programmers Workbench Library (**libPW.a**)

Syntax

```
char *regcmp (str [, str, . . . ], (char *) 0)
char *str, *str, . . . ;
```

```
char *regex (pat, subject [, ret, . . . ])
char *pat, *subject, *ret, . . . ;
```

```
extern char *--loc1;
```

Description

The **regcmp** subroutine compiles a regular expression (or pattern) and returns a pointer to the compiled form. The *str* parameters specify the pattern to be compiled. If more than one *str* parameter is given, then **regcmp** treats them as if they were concatenated together. It returns a **NULL** pointer if it encounters an incorrect parameter.

You can use the **regcmp** command to compile regular expressions into your C program, frequently eliminating the need to call the **regcmp** subroutine at run time.

The **regex** subroutine compares a compiled pattern to the *subject* string. Additional parameters are used to receive values. Upon successful completion, the **regex** subroutine returns a pointer to the next unmatched character. If the **regex** subroutine fails, a **NULL** pointer is returned. A global character pointer, **--loc1**, points to where the match began.

The **regcmp** and **regex** subroutines are borrowed from the **ed** command; however, the syntax and semantics have been changed slightly. You can use the following symbols with the **regcmp** and **regex** subroutines:

[] * . ^

These symbols have the same meaning as they do in the **ed** command.

- Within brackets the minus means "through." For example, [a-z] is equivalent to [abcd . . . xyz]. You can use the - by itself if the - is the last or first character. For example, the character class expression []-] matches the] (right bracket) and - (minus) characters.

\$ Matches the end of the string. Use \n to match a new-line character.

- + A regular expression followed by + means one or more times. For example, [0-9]+ is equivalent to [0-9][0-9]*.

{m} {m,} {m,u}

Integer values enclosed in { } indicate the number of times to apply the preceding regular expression. *m* is the minimum number and *u* is the maximum number. *u* must be less than 256. If you specify only *m*, it indicates the exact number of times to apply the regular expression. {m,} is equivalent to {m,∞} and matches *m* or more occurrences of the expression. The plus + (plus) and * (asterisk) operations are equivalent to {1,} and {0,}, respectively.

(. . .)\$*n*

This stores the value matched by the enclosed regular expression in the (*n* + 1)th *ret* parameter. Ten enclosed regular expressions are allowed. **regex** makes the assignments unconditionally.

(. . .)

Parentheses group subexpressions. An operator, such as *, +, or { } works on a single character or on a regular expression enclosed in parenthesis. For example, (a*(cb+))*\$0.

All of the above defined symbols are special. You must precede them with a \ (backslash) if you want to match the special symbol itself. For example, \\$ matches a dollar sign.

Note: **regcmp** uses the **malloc** subroutine to make the space for the vector. Always free the vectors that are not required. If you do not free the unrequired vectors, you may run out of memory if **regcmp** is called repeatedly. Use the following as a replacement for **malloc** to reuse the same vector, thus saving time and space:

```
/* . . . Your Program . . . */
```

```
malloc(n)
    int n;
{
    static int rebuf[256];

    return ((n <= sizeof(rebuf)) ? rebuf : NULL);
}
```


regcmp, regex

Examples

1. To perform a simple match:

```
char *cursor, *newcursor, *ptr;  
.  
.  
.  
newcursor = regex((ptr = regcmp("^\\n", 0)), cursor);  
free(ptr);
```

This matches a leading new-line character in the subject string pointed to by cursor.

2. To extract a substring that matches a pattern:

```
char ret0[9];  
char *newcursor, *name;  
.  
.  
.  
name = regcmp("[A-Za-z][A-Za-z0-9]{0,7}$", 0);  
newcursor = regex(name, "123Testing321", ret0);
```

This matches the eight-character identifier Testing3 and returns the address of the character after the last matched character (which is stored in newcursor). The string Testing3 is copied into the character array ret0.

Related Information

In this book: “regex: compile, step, advance” on page 3-277 and “malloc, free, realloc, calloc” on page 3-224.

The **ed** and **regcmp** commands in *AIX Operating System Commands Reference*.

regex: compile, step, advance

Purpose

Compiles and matches regular-expression patterns.

Library

None

Syntax

```
#define INIT           declarations
#define GETC( )       getc_code
#define PEEKC( )       peekc_code
#define UNGETC(c)      ungetc_code
#define RETURN(pointer) return_code
#define ERROR(val)     error_code
```

```
#include <regex.h>
```

```
char *compile (instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
char eof;
```

```
int step (string, expbuf)
char *string, *expbuf;
```

```
int advance (string, expbuf)
char *string, *expbuf;
```

Description

The **regex.h** header file defines several general-purpose subroutines that perform regular-expression pattern matching. Programs that perform regular-expression pattern matching such as **ed**, **sed**, **grep**, **bs**, and **expr** use this source file. In this way, only this file needs to be changed in order to maintain regular expression compatibility between programs.

The interface to this header file is complex. Programs that include this file define the following five macros before the **#include <regex.h>** statement. These macros are used by the **compile** subroutine.

INIT

This macro is used for dependent declarations and initializations. It is placed right after the declaration and opening { (left brace) of the **compile** subroutine. The definition of **INIT** must end with a ; (semicolon). **INIT** is frequently used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for **GETC**, **PEEKC**, and **UNGETC**. Otherwise, you can use **INIT** to declare external variables that **GETC**, **PEEKC**, and **UNGETC** need.

GETC()

This macro returns the value of the next character in the regular expression pattern. Successive calls to the **GETC** macro should return successive characters of the pattern.

PEEKC()

This macro returns the next character in the regular expression. Successive calls to the **PEEKC** macro should return the same character, which should also be the next character returned by the **GETC** macro.

UNGETC(c)

This macro causes the parameter *c* to be returned by the next call to the **GETC** and **PEEKC** macros. No more than one character of pushback is ever needed and this character is guaranteed to be that last character read by the **GETC** macro. The return value of the **UNGETC** macro is always ignored.

RETURN(pointer)

This macro is used on normal exit of the **compile** subroutine. The *pointer* parameter points to the first character immediately following the compiled regular expression. This is useful to programs that have memory allocation to manage.

ERROR(val)

This macro is used on abnormal exit from the **compile** subroutine. It should *never* contain a **return** statement. The *val* parameter is an error number. The error values and their meanings are:

Error	Meaning
11	Range endpoint too large.
16	Bad number.
25	\digit out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than two numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[] imbalance.

50 Regular expression overflow.

The **compile** subroutine compiles the regular expression for later use. The *instring* parameter is never used explicitly by the **compile** subroutine, but you can use it in your macros. For instance, you may want to pass the string containing the pattern as the *instring* parameter to **compile** and use the **INIT** macro to set a pointer to the beginning of this string. (The following example uses this technique.) If your macros do not use *instring*, then call **compile** with a value of `((char *) 0)` for this parameter.

The *expbuf* parameter points to a character array where the compiled regular expression is to be placed. The *endbuf* parameter points to the location that immediately follows the character array where the compiled regular expression is to be placed. If the compiled expression cannot fit in `(endbuf-expbuf)` bytes, the call **ERROR(50)** is made.

The *eof* parameter is the character that marks the end of the regular expression. For example, in **ed** this character is usually `'/'` (slash).

The **regexp.h** file defines other subroutines that perform actual regular-expression pattern matching. One of these is the **step** subroutine.

The *string* parameter of **step** is a pointer to a null-terminated string of characters to be checked for a match.

The *expbuf* parameter points to the compiled regular expression, which was obtained by a call to the **compile** subroutine.

The **step** subroutine returns the value 1 if the given string matches the pattern, and 0 if it does not match. If it matches, then **step** also sets two global character pointers: **loc1**, which points to the first character that matches the pattern, and **loc2**, which points to the character immediately following the last character that matches the pattern. Thus, if the regular expression matches the entire string, then **loc1** points to the first character of *string* and **loc2** points to the null character at the end of *string*.

The **step** subroutine uses the global variable **circf**, which is set by **compile** if the regular expression begins with a `^` (circumflex). If this variable is set, then **step** only tries to match the regular expression to the beginning of the string. If you compile more than one regular expression is before executing the first one, then save the value of **circf** for each compiled expression and set **circf** to that saved value before each call to **step**.

The **step** subroutine calls a subroutine named **advance** with the same parameters that it was passed. The **step** function increments through the *string* parameter and calls **advance** until **advance** returns a 1, indicating a match, or until the end of *string* is reached. To constrain *string* to the beginning of the string in all cases, call the **advance** subroutine directly instead of calling **step**.

When **advance** encounters an `*` (asterisk) or a `\{ \}` sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, **advance** backs up along the string until it finds a match or reaches the point in the string that initially matched the `*` or `\{ \}`. It is sometimes desirable to stop

regexp: compile, step, advance

this backing-up before the initial point in the string is reached. If the global character pointer **locs** is equal to the point in the string sometime during the backing up process, **advance** breaks out of the loop that backs up and returns 0. This is used by **ed** and **sed** for global substitutions on the whole line so that expressions like **s/y*/g** do not loop forever.

The **ecmp** and **getrange** subroutines are called by the routines previously mentioned.

Example

The following is an example of the regular expression macros and calls from the **grep** command.

```
#define INIT          register char *sp=instring;
#define GETC()        (*sp++)
#define PEEKC()        (*sp)
#define UNGETC(c)      (--sp)
#define RETURN(c)      return;
#define ERROR(c)       regerr()

#include <regexp.h>
. . .
compile (patstr, expbuf, &expbuf[ESIZE], '\0');
. . .
if (step (linebuf, expbuf))
    succeed ( );
. . .
```

Related Information

In this book: “**regcmp**, **regex**” on page 3-274.

The **ed**, **grep**, and **sed** commands in *AIX Operating System Commands Reference*.

scanf, fscanf, sscanf

Purpose

Converts formatted input.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int scanf (format [, pointer, . . . ])
char *format;
```

```
int fscanf (stream, format [, pointer, . . . ])
FILE *stream;
char *format;
```

```
int sscanf (s, format [, pointer, . . . ])
char *s, *format;
```

Description

The **scanf**, **fscanf**, and **sscanf** subroutines read character data, interpret it according to a format, and store the converted results into specified memory locations. These subroutines read their input from the following sources:

scanf reads from standard input (**stdin**).
fscanf reads from *stream*.
sscanf reads from the character string *s*.

The *format* parameter contains conversion specification used to interpret the input. The *pointer* parameters specify where to store the interpreted data.

The *format* parameter can contain the following:

- White space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described following, reads the input up to the next nonwhite space character. Unless there is a match in the control string, trailing white space (including a new-line character) is not read.
- Any character, except % (percent), which must match the next character of the input stream.

scanf, fscanf, sscanf

- A conversion specification that directs the conversion of the next input field. It consists of the following:
 1. The character % (percent)
 2. An optional assignment suppression character, * (asterisk)
 3. An optional numeric maximum field width.
 4. An optional l or h that indicates the size of the receiving variable.
 5. A conversion code.

The results from the conversion are placed in **pointer* unless you specify assignment suppression with *. Assignment suppression provides a way to describe an input field that is to be skipped. The input field is a string of nonwhite-space characters. It extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion code indicates how to interpret the input field. The corresponding *pointer* must usually be of a restricted type. You should not specify *pointer* for a suppressed field. You can use the following conversion codes:

- | | |
|-----------|---|
| % | A single % is expected in the input at this point; no assignment is done. |
| d | A decimal integer is expected; <i>pointer</i> should be an integer pointer. |
| u | An unsigned decimal integer is expected; <i>pointer</i> should be an unsigned integer pointer. |
| o | An octal integer is expected; <i>pointer</i> should be an integer pointer. |
| x | A hexadecimal integer is expected; <i>pointer</i> should be an integer pointer. |
| e, f, g | A floating-point number is expected. The next field is converted accordingly and stored through the corresponding parameter, which should be a pointer to a float . The input format for floating-point numbers is an optionally signed string of digits. It can contain a decimal point followed by an optional exponent field, which consists of an E or an e followed by an optionally signed integer. |
| s | A character string is expected. The <i>pointer</i> parameter should be a character pointer that points to an array of characters large enough to accept the string and ending with '\0'. The '\0' is added automatically. The input field ends with a white space character. |
| c | A character is expected. The <i>pointer</i> parameter should be a character pointer. The normal skip over white space is suppressed. Use %1s to read the next nonwhite-space character. If a field width is given, <i>pointer</i> should refer to a character array; the indicated number of characters is read. |
| [scanset] | Indicates string data and suppresses the normal skip over leading white space. The [is followed by a set of characters, called a <i>scanset</i> , and a]. The input field is the complete sequence of input characters that consists entirely of characters in the <i>scanset</i> . The ^ (circumflex), when it appears as the first character in <i>scanset</i> , serves as a complement operator and redefines the <i>scanset</i> as the set of |

all characters that are not contained in the remainder of the *scanset* string. Conventions used in the construction of the *scanset* follow:

- You can represent a range of characters by the construct *first-last*. Thus you can express [0123456789] as [0-9]. The *first* parameter must be lexically less than or equal to *last*, or else the - (dash) stands for itself. The - also stands for itself whenever it is the first or the last character in the *scanset*.
- You can include the] (right bracket), possibly preceded by a ^ (circumflex) as an element of the *scanset*, if it is the first character of the *scanset*. In this case it is not syntactically interpreted as the closing bracket. The corresponding *pointer* must point to a character array large enough to hold the data field and that ends with '\0'. The '\0' is added automatically.

The conversion characters **d**, **u**, **o**, and **x** can be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** can be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list.

The **scanf** subroutine conversion ends at the end-of-file, the end of the control string, or when an input character conflicts with the control string. If it ends with an input character conflict, the character that conflicts is not read from the input stream.

The **scanf** subroutine returns the number of successfully matched and assigned input items. This number can be 0 if there was an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

Unless there is a match in the control string, trailing white space (including a new-line character) is not read.

All of these subroutines return an **EOF** on the end of input and on a short count for missing or illegal data items.

The success of literal matches and suppressed assignments is not directly determinable.

Examples

1. To read several values and assign them to variables:

```
int i;
float x;
char name[50];

scanf ("%d%f%s", &i, &x, name);
with the input line:
25 54.32E-1 thompson
```


scanf, fscanf, sscanf

This assigns to `i` the value 25, to `x` the value 5.432, and to `name` the value `thompson\0`.

2. To perform simple pattern-matching while scanning the input:

```
int i;  
float x;  
char name[50];
```

```
scanf ("%2d%f%*d%[0-9]", &i, &x, name);
```

with the input:

```
56789 0123 56a72
```

This assigns 56 to `i`, 789.0 to `x`, skips 0123, and places the string `56\0` in `name`. The next call to **getchar** (see “`getc`, `fgetc`, `getchar`, `getw`” on page 3-189) returns `a`.

Related Information

In this book: “`atof`” on page 3-8, “`getc`, `fgetc`, `getchar`, `getw`” on page 3-189, “`printf`, `fprintf`, `sprintf`” on page 3-262, “`stdio`” on page 3-297, and “`strtol`, `atol`, `atoi`” on page 3-302.

Examples of using **scanf** in *C Language Guide and Reference*.

setbuf

Purpose

Assigns buffering to a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;
```

Description

The **setbuf** subroutine causes the character array pointed to by the *buf* parameter to be used instead of an automatically allocated buffer. Use the **setbuf** subroutine after a stream has been opened but before it is read or written.

If the *buf* parameter is a **NULL** character pointer, input/output is completely unbuffered.

A constant, **BUFSIZ**, defined in the **stdio.h** header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from the **malloc** subroutine at the time of the first **getc** or **putc** on the file, except that the standard error stream, **stderr**, is normally not buffered.

Output streams directed to terminals are always either line-buffered or unbuffered.

Note: A common source of error is allocating buffer space as an automatic variable in a code block, and then failing to close the stream in the same block.

Related Information

In this book: “fopen, freopen, fdopen” on page 3-157, “getc, fgetc, getchar, getw” on page 3-189, “malloc, free, realloc, calloc” on page 3-224, “putc, putchar, fputc, putw” on page 3-266, and “stdio” on page 3-297.

setjmp, longjmp

Purpose

Saves and restores the current execution context.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <setjmp.h>
```

```
int setjmp (ctxt)  
jmp_buf ctxt;
```

```
void longjmp (ctxt, val)  
jmp_buf ctxt;  
int val;
```

Description

The **setjmp** and **longjmp** subroutines can be useful when handling errors and interrupts encountered in low-level subroutines of a program.

The **setjmp** subroutine saves the current stack context and signal mask in the buffer specified by the *ctxt* parameter. The **setjmp** subroutine returns a value of 0.

The **longjmp** subroutine restores the stack context and signal mask that were saved by the **setjmp** subroutine in the corresponding *ctxt* buffer. After the **longjmp** subroutine has completed, the program execution continues as if the corresponding call to **setjmp** had just returned the value of the *val* parameter. The subroutine that called **setjmp** must not have returned before the completion of the **longjmp** subroutine.

The **longjmp** subroutine cannot return 0 to the previous context. The value 0 is reserved to indicate the actual return from the **setjmp** subroutine when first called by the program. If the **longjmp** subroutine is passed a *val* parameter of 0, then execution continues as if the corresponding call to the **setjmp** subroutine had returned a value of 1. All accessible data have values as of the time the **longjmp** subroutine is called.

Warning: If the **longjmp** subroutine is called with a *ctxt* parameter that was not previously set by **setjmp**, or if the subroutine that made the corresponding call to **setjmp** has already returned, then the results of the **longjmp** subroutine are undefined.

setjmp, longjmp

Related Information

In this book: “signal” on page 2-139 and “sigvec” on page 2-150.

sgetl, sputl

Purpose

Accesses long numeric data in a machine-independent fashion.

Library

Object File Access Routine Library (**libld.a**)

Syntax

long sgetl (*buffer*)
char **buffer*;

void sputl (*value*, *buffer*)
long *value*;
char **buffer*;

Description

The **sgetl** subroutine retrieves 4 bytes from memory starting at the location pointed to by the **buffer** parameter. It then returns the bytes as a **long** value with the byte ordering of the host machine.

The **sputl** subroutine stores the 4 bytes of the *value* parameter into memory starting at the location pointed to by the *buffer* parameter. The order of the bytes is the same across all machines.

Using **sputl** and **sgetl** subroutines together provides a machine-independent way of storing long numeric data in an ASCII file. For example, the numeric data stored in the portable archive file format is accessed with the **sputl** and **sgetl** subroutines.

Related Information

In this book: “frexp, ldexp, modf” on page 3-179, and “ar” on page 4-18.

sin, cos, tan, asin, acos, atan, atan2

sin, cos, tan, asin, acos, atan, atan2

Purpose

Computes trigonometric functions.

Library

Math Library (**libm.a**)

Syntax

#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double tan (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double x, y;

Description

The **sin**, **cos**, and **tan** subroutines return the sine, cosine and tangent, respectively, of their parameters, which are in radians.

The **asin** subroutine returns the arcsine of x , in the range $-\pi/2$ to $\pi/2$.

The **acos** subroutine returns the arccosine of x , in the range 0 to π .

The **atan** subroutine returns the arctangent of x , in the range $-\pi/2$ to $\pi/2$.

The **atan2** subroutine returns the arctangent of y/x , in the range $-\pi$ to π , using the signs of both parameters to determine the quadrant of the return value.

The **sin**, **cos** and **tan** subroutines lose accuracy when passed a large value for the x parameter. For sufficiently large parameters, these functions return 0 when there would otherwise be a complete loss of significance. In this case, a message that indicates a

TLOSS error is written to standard error. For less extreme values, a **PLOSS** error is generated but no message is written. In both cases, **errno** is set to **ERANGE**.

The **tan** subroutine returns **HUGE** if its parameter is near an odd multiple of $\pi/2$ when the correct value would overflow, and sets **errno** to **ERANGE**.

The **asin** and **acos** subroutines return 0 and set **errno** to **EDOM** if their parameters are larger than 1.0. In addition, an error message that indicates a domain error is written to the standard error output.

You can change the error-handling procedures with the **matherr** subroutine.

Related Information

In this book: “**matherr**” on page 3-226.

sinh, cosh, tanh

sinh, cosh, tanh

Purpose

Computes hyperbolic functions.

Library

Math Library (**libm.a**)

Syntax

```
#include <math.h>
```

```
double sinh (x)  
double x;
```

```
double tanh (x)  
double x;
```

```
double cosh (x)  
double x;
```

Description

The **sinh** subroutine returns the hyperbolic sine of its parameter. The **cosh** subroutine returns the hyperbolic cosine of its parameter. The **tanh** subroutine returns the hyperbolic tangent of its parameter.

The **sinh** and the **cosh** subroutines return **HUGE** if the correct value overflows. **errno** is also set to **ERANGE**.

You can use the **matherr** subroutine to change these error-handling procedures. See “**matherr**” on page 3-226 for details.

sleep

Purpose

Suspends execution of the current process for an interval of time.

Library

Standard C Library (**libc.a**)

Syntax

```
unsigned int sleep (seconds)  
unsigned int seconds;
```

Description

The **sleep** subroutine causes the current process to suspend execution for the number of seconds specified by the *seconds* parameter. The sleep routine sets an alarm and pauses until that alarm or some other signal occurs.

The actual sleep time of the process may be either shorter or longer than the requested sleep time. The sleep time may be shorter because:

- Wakeups occur on the second at fixed 1-second intervals according to an internal clock.
- Any caught signal terminates the sleep following execution of that signal's catching routine.

The sleep time may be longer than the requested sleep time due to the scheduling of other activities in the system.

The value returned by the **sleep** subroutine is the requested sleep time minus the time actually slept.

The process calling the **sleep** subroutine may have set an alarm prior to calling the **sleep** subroutine.

If a previous alarm has been set, and the **sleep** subroutine's sleep time exceeds the process's previously set sleep time, the process only sleeps until the time specified by the previously set alarm and the calling process's alarm catch routine is executed just before the **sleep** subroutine returns.

sleep

If a previous alarm has been set, and the **sleep** subroutine's sleep time is less than the process's previously set sleep time, the current process is suspended from execution for the number of seconds specified by the **sleep** subroutine. The previously set alarm is reset to go off at the same time it would have without the **sleep** subroutines intervention.

Warning: The results are undefined if, while it is sleeping, the calling program issues any other **alarm** or **sleep** calls. This can happen if a signal arrives in the interim and the signal handler calls **alarm** or **sleep**.

Related Information

In this book: “alarm” on page 2-13, “pause” on page 2-92, and “signal” on page 2-139.

ssignal, gsignal

Purpose

Implements a software signal facility.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <signal.h>
```

```
int (*ssignal (sig, action)) ( )
int sig, (*action) ( );
```

```
int gsignal (sig)
int sig;
```

Description

The **ssignal** and **gsignal** subroutines implement a software facility similar to that of the **signal** and **kill** system calls. However, there is no connection between the two facilities. User programs can use **ssignal** and **gsignal** to handle exceptional processing within an application. **signal** and related system calls handle system-defined exceptions.

The software signals available are associated with integers in the range 1 through 15. Other values are reserved for use by the C library and should not be used.

The **ssignal** subroutine associates the procedure specified by the *action* parameter with the software signal specified by the *sig* parameter. The **gsignal** subroutine “raises” the signal *sig*, causing the procedure specified by the *action* parameter to be taken.

The *action* parameter is either a pointer to a user-defined subroutine, or one of the constants **SIG_DFL** (default action) and **SIG_IGN** (ignore signal). The **ssignal** subroutine returns the procedure that was previously established for that signal. If no procedure was established before, or if the signal number is illegal, then **ssignal** returns the value **SIG_DFL**.

ssignal, gsignal

The **gsignal** subroutine “raises” the signal specified by the *sig* parameter by doing the following:

- If the procedure for *sig* is **SIG_DFL**, then the **gsignal** subroutine returns a value of 0 and takes no other action.
- If the procedure for *sig* is **SIG_IGN**, then the **gsignal** subroutine returns a value of 1 and takes no other action.
- If the procedure for *sig* is a subroutine, then the *action* value is reset to **SIG_DFL** and the subroutine is called with *sig* passed as its parameter. The **gsignal** subroutine returns the value that is returned by the signal-handling subroutine.
- If the procedure for *sig* is an illegal value or if no procedure was ever specified for that signal, then **gsignal** returns a value of 0 and takes no other action.

Related Information

In this book: “kill” on page 2-58 and “signal” on page 2-139.

stdio

Purpose

Performs standard buffered input and output operations.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
FILE *stdin, *stdout, *stderr;
```

Description

These macros and subroutines provide an efficient user-level I/O buffering scheme.

The in-line macros **getc** and **putc** handle characters quickly. The following macros and subroutines all use the **getc** and **putc** macros:

getchar macro
putchar macro
fgetc subroutine
fgets subroutine
fprintf subroutine
fputc subroutine
fputs subroutine
fread subroutine

fscanf subroutine
fwrite subroutine
gets subroutine
getw subroutine
printf subroutine
puts subroutine
putw subroutine
scanf subroutine

A file with associated buffering is called a **stream** and is declared to be a pointer to the defined type **FILE**. The **fopen** subroutine constructs descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **stdio.h** header file and associated with the standard open streams:

stdin Standard input stream
stdout Standard output stream
stderr Standard error output stream.

The constant **NULL** (0) designates a special pointer value that does not point to any data structure.

Most integer subroutines that deal with streams return the constant **EOF** (-1) upon end-of-file or an error. See each individual subroutine for detailed information about the return value.

Programs that use this input/output package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The subroutines and constants in the input/output package are declared in the header file and do not need any further declaration. The constants and the following routines are implemented as macros. Redclaration of these names is not allowed.

getc
getchar
putc
putchar

feof
ferror
clearerr
fileno

Warning: Invalid stream pointers usually cause errors, possibly including program termination. Individual subroutine descriptions describe the possible error conditions.

Related Information

In this book: “open” on page 2-88, “close” on page 2-25, “lseek” on page 2-65, “pipe” on page 2-93, “read, readx” on page 2-104, “write, writex” on page 2-178, “ctermid” on page 3-41, “cuserid” on page 3-62, “fclose, fflush” on page 3-152, “feof, ferror, clearerr, fileno” on page 3-154, “fopen, freopen, fdopen” on page 3-157, “fread, fwrite” on page 3-177, “fseek, rewind, ftell” on page 3-181, “getc, fgetc, getchar, getw” on page 3-189, “gets, fgets” on page 3-205, “popen, pclose” on page 3-260, “printf, fprintf, sprintf” on page 3-262, “putc, putchar, fputc, putw” on page 3-266, “puts, fputs” on page 3-269, “scanf, fscanf, sscanf” on page 3-281, “setbuf” on page 3-285, “system” on page 3-305, “tmpfile” on page 3-307, “tmpnam, tempnam” on page 3-308, and “ungetc” on page 3-320.

string

Purpose

Performs operations on strings.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>
```

```
char *strcat (s1, s2)  
char *s1, *s2;
```

```
char *strncat (s1, s2, n)  
char *s1, *s2;  
int n;
```

```
int strcmp (s1, s2)  
char *s1, *s2;
```

```
int strncmp (s1, s2, n)  
char *s1, *s2;  
int n;
```

```
char *strcpy (s1, s2)  
char *s1, *s2;
```

```
char *strncpy (s1, s2, n)  
char *s1, *s2;  
int n;
```

```
int strlen (s)  
char *s;
```

```
char *strchr (s, c)  
char *s, c;
```

```
char *strrchr (s, c)  
char *s, c;
```

```
char *strpbrk (s1, s2)  
char *s1, *s2;
```

```
int strspn (s1, s2)  
char *s1, *s2;
```

```
int strcspn (s1, s2)  
char *s1, *s2;
```

```
char *strtok (s1, s2)  
char *s1, *s2;
```


Description

The **string** subroutines copy, compare, and append strings in memory, and they determine such things as location, size, and existence of strings in memory.

The parameters *s1*, *s2* and *s* point to strings. A string is an array of characters terminated by a null character. The subroutines **strcat**, **strncat**, **strcpy**, and **strncpy** all alter *s1*. They do not check for overflow of the array pointed to by *s1*. All string movement is performed character by character and starts at the left. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **string.h** header file.

The **strcat** subroutine adds a copy of the string pointed to by the *s2* parameter to the end of the string pointed to by the *s1* parameter. The **strcat** subroutine returns a pointer to the null-terminated result.

The **strncat** subroutine copies at most *n* bytes of *s2* to the end of the string pointed to by the *s1* parameter. Copying stops before *n* bytes if a null character is encountered in the *s2* string. The **strncat** subroutine returns a pointer to the null-terminated result.

The **strcmp** subroutine lexicographically compares the string pointed to by the *s1* parameter to the string pointed to by the *s2* parameter. The **strcmp** subroutine uses native character comparison, which may be signed or unsigned. The **strcmp** subroutine returns a value that is:

Less than 0	If <i>s1</i> is less than <i>s2</i>
Equal to 0	If <i>s1</i> is equal to <i>s2</i>
Greater than 0	If <i>s1</i> is greater than <i>s2</i> .

The **strncmp** subroutine makes the same comparison as **strcmp**, but it compares at most *n* pairs of characters.

The **strcpy** subroutine copies the string pointed to by the *s2* parameter to the character array pointed to by the *s1* parameter. Copying stops when the null character is copied. The **strcpy** subroutine returns the value of the *s1* parameter.

The **strncpy** subroutine copies *n* bytes from the string pointed to by the *s2* parameter to the character array pointed to by the *s1* parameter. If *s2* is less than *n* characters long, then **strncpy** pads *s1* with trailing null characters to fill *n* bytes. If *s2* is *n* or more characters long, then only the first *n* characters are copied and the result is not terminated with a null character. The **strncpy** subroutine returns the value of the *s1* parameter.

The **strlen** subroutine returns the number of characters in the string pointed to by the *s* parameter, not including the terminating null character.

The **strchr** subroutine returns a pointer to the first occurrence of the character specified by the *c* parameter in the string pointed to by the *s* parameter. A **NULL** pointer is returned if the character does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **strrchr** subroutine returns a pointer to the last occurrence of the character specified by the *c* parameter in the string pointed to by the *s* parameter. A **NULL** pointer is returned if the character does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **strpbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the *s1* parameter of any character from the string pointed to by the *s2* parameter. A **NULL** pointer is returned if no character matches.

The **strspn** subroutine returns the length of the initial segment of the string pointed to by the *s1* parameter that consists entirely of characters from the string pointed to by the *s2* parameter.

The **strcspn** subroutine returns the length of the initial segment of the string pointed to by the *s1* parameter that consists entirely of characters *not* from the string pointed to by the *s2* parameter.

The **strtok** subroutine returns a pointer to an occurrence of a text token in the string pointed to by the *s1* parameter. The *s2* parameter specifies a set of token delimiters. If the *s1* parameter is anything other than **NULL**, then the **strtok** subroutine reads the string pointed to by the *s1* parameter until it finds one of the delimiter characters specified by the *s2* parameter. It then stores a null character into the string, replacing the delimiter, and returns a pointer to the first character of the text token. The **strtok** subroutine keeps track of its position in the string so that subsequent calls with a **NULL** *s1* parameter step through the string. The delimiters specified by the *s2* parameter can be changed for subsequent calls to **strtok**. When no tokens remain in the string pointed to by the *s1* parameter, the **strtok** subroutine returns a **NULL** pointer.

Related Information

In this book: “memccpy, memchr, memcmp, memcpy, memset” on page 3-233 and “swab” on page 3-304.

strtol, atol, atoi

strtol, atol, atoi

Purpose

Converts a string to an integer.

Library

Standard C Library (**libc.a**)

Syntax

```
long strtol (str, ptr, base)  
char *str, **ptr;  
int base;
```

```
long atol (str)  
char *str;
```

```
int atoi (str)  
char *str;
```

Description

The **strtol** subroutine returns a long integer whose value is represented by the character string *str*. **strtol** scans the string up to the first character that is inconsistent with the *base*. Leading white-space characters are ignored.

Warning: Overflow conditions are ignored.

If the value of *ptr* is not (**char ****) **NULL**, then a pointer to the character that terminated the scan is stored in **ptr*. If an integer cannot be formed, **ptr* is set to *str*, and 0 is returned.

If the *base* parameter is positive and not greater than 36, then it is used as the base for conversion. After an optional leading sign, leading zeroes are ignored. 0x or 0X is ignored if *base* is 16.

If the *base* parameter is 0, the string determines the base. Thus, after an optional leading sign, a leading 0 indicates octal conversion, and a leading 0x or 0X indicates hexadecimal conversion. The default is to use decimal conversion.

Note: Truncation from **long** to **int** can take place upon assignment, or by an explicit cast.

The **atol** (*str*) subroutine call is equivalent to **strtol** (*str*, (**char ****) **NULL**, **10**).

The **atoi** (*str*) subroutine call is equivalent to (**int**) **strtol** (*str*, (**char ****) **NULL**, **10**).

The **atoi** and **atol** subroutines do not actually call **strtol**.

Related Information

In this book: “**atof**” on page 3-8 and “**scanf**, **fscanf**, **sscanf**” on page 3-281.

swab

swab

Purpose

Copies bytes.

Library

Standard C Library (**libc.a**)

Syntax

```
void swab (from, to, nbytes)  
char *from, *to;  
int nbytes;
```

Description

The **swab** subroutine copies *nbytes* bytes from the location pointed to by the *from* parameter to the array pointed to by the *to* parameter, exchanging adjacent even and odd bytes.

The *nbytes* parameter should be even and nonnegative. If the *nbytes* parameter is odd and positive, the **swab** uses *nbytes*-1 instead. If the *nbytes* parameter is negative, then **swab** does nothing.

Related Information

In this book: “memccpy, memchr, memcmp, memcpy, memset” on page 3-233 and “string” on page 3-299.

system

Purpose

Runs a shell command.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int system (string)  
char *string;
```

Description

The **system** subroutine passes the *string* parameter to the **sh** command as input. Then **sh** interprets *string* as a command and runs it.

The **system** subroutine invokes the **fork** system call to create a child process that in turn uses **exec** to run **/bin/sh**, which interprets the shell command contained in the *string* parameter. The current process waits until the shell has completed, then returns the exit status of the shell.

Note: The **system** subroutine runs only **sh** shell commands (also called *Bourne shell* commands). The results are unpredictable if the *string* parameter is not a valid **sh** shell command.

Return Value

Upon successful completion, the **system** subroutine returns the exit status of the shell. See “wait” on page 2-176 for an explanation of the exit status.

If the **fork** fails, then **system** returns a value of -1. If the **exec** fails, then it returns 127. In either case, **errno** is set to indicate the error.

File

/bin/sh

Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “exit, _exit” on page 2-38, “fork” on page 2-44, and “wait” on page 2-176.

The **sh** command in *AIX Operating System Commands Reference*.

tmpfile

Purpose

Creates a temporary file.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
FILE *tmpfile ( )
```

Description

The **tmpfile** subroutine creates a temporary file and returns its **FILE** pointer. The file is opened for update. The temporary file is automatically deleted when the process using it terminates.

If the file cannot be opened, **tmpfile** writes an error message to the standard error output and returns a **NULL** pointer.

Related Information

In this book: “creat” on page 2-27, “unlink” on page 2-168, “fopen, freopen, fdopen” on page 3-157, “mktemp” on page 3-235, “stdio” on page 3-297, and “tmpnam, tempnam” on page 3-308.

tmpnam, tmpnam

tmpnam, tmpnam

Purpose

Constructs the name for a temporary file.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
char *tmpnam (s)  
char *s;
```

```
char *tempnam (dir, pfx)  
char *dir, *pfx;
```

Description

The **tmpnam** and **tempnam** subroutines generate file names for temporary files.

The **tmpnam** subroutine generates a file name using the path name defined as **P_tmpdir** in the **stdio.h** header file. If the *s* parameter is **NULL**, the **tmpnam** subroutine places its result into an internal static area and returns a pointer to that area. The next call to this subroutine destroys the contents of the area.

If the *s* parameter is not **NULL**, it is assumed to be the address of an array of at least the number of bytes specified by **L_tmpnam**. **L_tmpnam** is a constant defined in **stdio.h**. The **tmpnam** subroutine places its results into that array and returns the value of the *s* parameter.

The **tempnam** subroutine allows you to control the choice of a directory. The *dir* parameter points to the path name of the directory in which the file is to be created. If the *dir* parameter is **NULL** or points to a string which is not a path name for an appropriate directory, the path name defined as **P_tmpdir** in the **stdio.h** header file is used. If that path name is not accessible, **/tmp** is used. You can bypass the selection of a path name by providing an environment variable, **TMPDIR**, in the user's environment. The value of the **TMPDIR** variable is a path name for the desired temporary-file directory. If the **TMPDIR** variable is used, both the *dir* parameter and **L_tmpnam** are ignored.

The *pfx* parameter of the **tempname** subroutine allows you to specify an initial character sequence with which the file name begins. The *pfx* parameter can be **NULL**, or it can

point to a string of up to five characters to be used as the first few characters of the temporary file name.

The **tempnam** subroutine uses the **malloc** subroutine to obtain space for the constructed file name. The return value is a pointer to this space. Therefore, the pointer value returned by **tempnam** can be used as a parameter to the **free** subroutine.

If the **tempnam** subroutine cannot return the expected result for any reason (for example, if the **malloc** subroutine fails, or if an appropriate directory cannot be found), then it returns a **NULL** pointer.

Warning: The **tmpnam** and **tempnam** subroutines generate a different file name each time they are called. If they are called more than 17,576 times by a single process, they start recycling previously used names.

Files created using these subroutines reside in a directory intended for temporary use, and their names are unique. It is your responsibility to use the **unlink** system call to remove the file when no longer needed.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This should not happen if that other process uses these subroutines or the **mktemp** subroutine, and if the file names are chosen to make duplication by other means unlikely.

Related Information

In this book: “creat” on page 2-27, “unlink” on page 2-168, “fopen, freopen, fdopen” on page 3-157, “malloc, free, realloc, calloc” on page 3-224, “mktemp” on page 3-235, “tmpfile” on page 3-307, and “environment” on page 5-44.

toupper, tolower, _toupper, _tolower, toascii

toupper, tolower, _toupper, _tolower, toascii

Purpose

Translates characters.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <ctype.h>
```

```
int toupper (c)  
int c;
```

```
int _toupper (c)  
int c;
```

```
int tolower (c)  
int c;
```

```
int _tolower (c)  
int c;
```

```
int toascii (c)  
int c;
```

Description

The **toupper** and the **tolower** subroutines have as domain the range of the **getc** subroutine: from -1 through 255.

If the parameter of the **toupper** subroutine represents a lowercase letter, the result is the corresponding uppercase letter. If the parameter of the **tolower** subroutine represents an uppercase letter, the result is the corresponding lowercase letter. All other values in the domain are returned unchanged.

The **_toupper** and **_tolower** routines are macros that accomplish the same thing as **toupper** and **tolower**, but they have restricted domains and they are faster. **_toupper** requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. **_tolower** requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Arguments outside the domain cause undefined results.

The **toascii** subroutine yields the value of its parameter with all bits that are not part of a standard ASCII character turned off. It is intended for compatibility with other systems.

Related Information

In this book: “ctype” on page 3-46 and “getc, fgetc, getchar, getw” on page 3-189.

trace_on

trace_on

Purpose

Checks whether trace channel is enabled.

Library

Run-time Services Library (**librts.a**)

Syntax

```
int trace_on(chanmask)  
unsigned long chanmask;
```

Description

The **trace_on** subroutine queries the application trace device driver to determine whether a given trace channel is enabled. **trace_on** allows a program to avoid the unnecessary overhead of setting up the trace message when its trace channel is disabled. **trace_on** is a C run-time subroutine and should be used by application programs, but not by device drivers.

The *chanmask* parameter is a mask with the bit corresponding to the channel number set. It can be formed by the expression $(1 \ll 31 - \text{channum})$. User programs can use only channel number 31, which means that the value of *chanmask* must be 1 for user programs.

Making repeated calls to the trace device driver involves significant overhead, so call **trace_on** only once: either at the start of processing or just before the first trace point in the program.

If the application trace device driver is not already open, **trace_on** opens it.

Upon successful completion, **trace_on** returns 1 if the channel is enabled, or 0 if the channel is disabled. If the **trace_on** subroutine fails, a message is written to the standard error output, and a value of -1 is returned.

File

/dev/apltrace

Related Information

In this book: “trcunix” on page 3-314, “trace” on page 6-117, and “trsave” on page E-6.

The **trace** command in *AIX Operating System Commands Reference*.

The discussion of trace in *AIX Operating System Programming Tools and Interfaces*.

trcunix

trcunix

Purpose

Records application trace log entries.

Library

Run-time Services Library (**librts.a**)

Syntax

```
int trcunix (buf, cnt)
char *buf;
unsigned int cnt;
```

Description

The **trcunix** subroutine invokes the application trace device driver to record a trace log entry. **trcunix** is a C run-time subroutine. Device drivers should use the **trsave** subroutine to log trace events.

The *buf* parameter points to a buffer containing a 2-byte *traceid* followed by up to 20 bytes of user-defined trace data. The high-order 5 bits of the *traceid* specify the channel number, and the low-order 11 bits specify the hookid for the message. User programs may use only channel number 31. The *cnt* parameter specifies the number of bytes in the buffer, including the *traceid*.

If the application trace device driver is not open, then **trcunix** opens it before writing the trace log entry to it.

Return Value

Upon successful completion, a value of 0 is returned and a trace log entry is written to **/dev/appltrace**. If the **trcunix** subroutine fails, an error message is written to the standard error output, and a value of -1 is returned.

File

`/dev/appltrace`

Related Information

In this book: “trace_on” on page 3-312, “trace” on page 6-117, and “trsave” on page E-6.

The **trace** command in *AIX Operating System Commands Reference*.

The discussion of trace in *AIX Operating System Programming Tools and Interfaces*.

tsearch, tdelete, twalk

tsearch, tdelete, twalk

Purpose

Manages binary search trees.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar) ( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar) ( );

void twalk ((char *) root, action)
void (*action) ( );
```

Description

The **tsearch** subroutine performs a binary tree search. The algorithm is generalized from Donald E. Knuth's *The Art of Computer Programming*, Volume 3, 6.2.2, Algorithm T.* It returns a pointer into a tree indicating where the data specified by the *key* parameter can be found. If the data specified by the *key* parameter is not found, the data is added to the tree in the correct place. If there is not enough space available to create a new node, a **NULL** pointer is returned. The *rootp* parameter points to a variable that points to the root of the tree. If the *rootp* parameter is **NULL**, the variable is set to point to the root of a new tree.

The *compar* parameter is a pointer to the comparison function, which is called with two parameters that point to the elements being compared. The comparison function must compare its parameters and return a value as follows:

* Reading, Massachusetts: Addison-Wesley, 1981.

- If the first parameter is less than the second parameter, *compar* must return a value less than 0.
- If the first parameter is equal to the second parameter, *compar* must return 0.
- If the first parameter is greater than the second parameter, *compar* must return a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

If the *rootp* parameter is **NULL** on entry, then a **NULL** pointer is returned.

The **tdelete** subroutine deletes the data specified by the *key* parameter. It is generalized from Knuth (6.2.2) Algorithm D. The *rootp* and *compar* parameters perform the same function as they do for the **tsearch** subroutine. The variable pointed to by the *rootp* parameter will be changed if the deleted node is the root of the binary tree. The **tdelete** subroutine returns a pointer to the parent node of the deleted node. If the data is not found, a **NULL** pointer is returned. If the *rootp* parameter is **NULL** on entry, then a **NULL** pointer is returned.

The **twalk** subroutine steps through the binary search tree whose root is pointed to by the *root* parameter. (Any node in a tree can be used as the root to step through the tree below that node.) The *action* parameter is the name of a routine to be invoked at each node. The routine specified by the *action* parameter is called with three parameters. The first parameter is the address of the node currently being pointed to. The second parameter is a value from an enumeration data type

```
typedef enum {preorder, postorder, endorder, leaf} VISIT;
```

(This data type is defined in the **search.h** header file). The actual value of the second parameter depends on whether this is the first, second, or third time that the node has been visited during a depth-first, left-to-right traversal of the tree, or whether the node is a *leaf*. A leaf is a node that is not the parent of another node. The third parameter is the level of the node in the tree, with the root node being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element and cast to type pointer-to-character. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Related Information

In this book: “bsearch” on page 3-11, “hsearch, hcreate, hdestroy” on page 3-211, and “lsearch” on page 3-222.

ttyname, isatty

ttyname, isatty

Purpose

Gets the name of a terminal.

Library

Standard C Library (**libc.a**)

Syntax

char *ttyname (*fildes*)
int fildes;

int isatty (*fildes*)
int fildes;

Description

The **ttyname** subroutine gets the name of a terminal. It returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor specified by the *fildes* parameter. A **NULL** pointer is returned if the file descriptor does not describe a terminal device in directory **/dev**.

The **isatty** subroutine determines if the device associated with the file descriptor specified by the *fildes* parameter is a terminal. If the specified file descriptor is associated with a terminal, the **isatty** subroutine returns a value of 1. If the file descriptor is not associated with a terminal, a value of 0 is returned.

The return value of **ttyname** points to static data whose contents are overwritten by each call.

Files

/dev/*

ttyslot

Purpose

Finds the slot in the **utmp** file for the current user.

Library

Standard C Library (**libc.a**)

Syntax

```
int ttyslot ( )
```

Description

The **ttyslot** subroutine returns the index of the current user's entry in the **/etc/utmp** file. The **ttyslot** subroutine scans the **/etc/utmp** file for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1, or 2).

The **ttyslot** subroutine returns 0 if an error was encountered while searching for the terminal name, or if none of the first three file descriptors (0, 1, and 2) is associated with a terminal device.

Files

/etc/inittab
/etc/utmp

Related Information

In this book: “getutent” on page 3-208 and “ttyname, isatty” on page 3-318.

ungetc

ungetc

Purpose

Pushes a character back into input stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>

int ungetc (c, stream)
int c;
FILE *stream;
```

Description

The **ungetc** subroutine inserts the character specified by the *c* parameter into the buffer associated with the input stream specified by the *stream* parameter. This causes the next call to the **getc** subroutine to return *c*. **ungetc** returns *c*, and leaves the *stream* file unchanged.

If the *c* parameter is **EOF**, then the **ungetc** subroutine does not place anything in the buffer and a value of **EOF** is returned.

You can always push one character back onto a stream, provided that something has been read from the stream or **setbuf** has been called. The **fseek** subroutine erases all memory of inserted characters.

The **ungetc** subroutine returns **EOF** if it cannot insert the character.

Related Information

In this book: “fseek, rewind, ftell” on page 3-181, “getc, fgetc, getchar, getw” on page 3-189, “setbuf” on page 3-285 and “stdio” on page 3-297.

varargs

varargs

Purpose

Handles a variable-length parameter list.

Syntax

#include <varargs.h>

va_alist

type va_arg (argp, type)

va_list argp;

va_dcl

void va_end (argp)

void va_start (argp)
va_list argp;

va_list argp;

Description

This set of macros allows you to write portable subroutines that accept a variable number of parameters. Subroutines that have variable-length parameter lists (such as **printf**), but that do not use **varargs**, are inherently nonportable because different systems use different parameter-passing conventions.

va_alist Is used as the parameter list in the function header.

va_dcl Is the declaration for **va_alist**. No semicolon should follow **va_dcl**.

va_list Defines the type of the variable used to traverse the list.

va_start Initializes *argp* to point to the beginning of the list.

argp Is a variable that the **varargs** macros use to keep track of the current location in the parameter list. Do not modify this variable.

va_arg Returns the next parameter in the list pointed to by *argp*. *type* is the data type that the parameter is expected to be. Different types can be mixed, but your subroutine must know what type of parameter is expected because it cannot be determined at runtime. The **printf** subroutine solves this problem by using its *format* parameter to determine the parameter types expected.

var_end Cleans up at the end.

Your subroutine can traverse, or scan, the parameter list more than once. Start each traversal with a call to **va_start** and end it with **var_end**.

Note: The calling routine is responsible for specifying the number of parameters because it is not always possible to determine this from the stack frame. For example, **execl** is passed a **NULL** pointer to signal the end of the list. **printf** determines the number of parameters from its *format* parameter.

Specifying **char**, **short**, or **float** as the second parameter to **va_arg** is not portable because parameters seen by the called subroutine are not **char**, **short**, or **float**. The C compiler converts **char** and **short** parameters to **int**, and it converts **float** parameters to **double** before passing them to a subroutine.

Example

The following example is a possible implementation of **execl** system call:

```
#include <varargs.h>

#define MAXARGS 100
/*
** execl is called by
** execl(file, arg1, arg2, . . . , (char *) 0);
*/
execl(va_alist)
    va_dcl
{
    va_list ap;
    char *file;
    char *args[MAXARGS];
    int argno = 0;

    va_start(ap);
    file = va_arg(ap, char *);
    while ((args[argno++] = va_arg(ap, char *)) != (char *) 0)
        ; /* Empty loop body */
    va_end(ap);
    return (execv(file, args));
}
```


Related Information

In this book: “exec: execl, execv, execl, execve, execlp, execvp” on page 2-32, “printf, fprintf, sprintf” on page 3-262, and “vprintf, vfprintf, vsprintf” on page 3-325.

vprintf, vfprintf, vsprintf

Purpose

Formats a **varargs** parameter list for output.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
#include <varargs.h>
```

```
int vprintf (format, argp)
char *format;
va_list argp;
```

```
int vfprintf (stream, format, argp)
FILE *stream;
```

```
char *format;
va_list argp;
```

```
int vsprintf (s, format, argp)
char *s, *format;
va_list argp;
```

Description

The **vprintf**, **vfprintf**, and **vsprintf** subroutines format and write **varargs** parameter lists. They are the same as the **printf**, **fprintf**, and **sprintf** subroutines, respectively, except that they are not called with a variable number of parameters. Instead, they are called with a parameter list pointer as defined by “varargs” on page 3-322.

Example

The following example demonstrates how the **vfprintf** subroutine could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
```

```
/* error should be called with the syntax: */
/* error(routine-name, format [, value, . . . ]); */
```

vprintf, vfprintf, vsprintf

```
/*VARARGS*/

void error(va_alist)
va_dcl
/*
** Note that the function name and format arguments
** cannot be separately declared because of the
** definition of varargs.
*/
{
    va_list args;
    char *fmt;

    va_start(args);
    /*
    ** Display the name of the function that called error
    */
    (void) fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
    /*
    ** Display the remainder of the message
    */
    fmt = va_arg(args, char *);
    (void) vfprintf(fmt, args);
    va_end(args);
    (void) abort();
}
```

Related Information

In this book: "printf, fprintf, sprintf" on page 3-262.

vrcppr

Purpose

Installs or removes a protocol procedure.

Library

Run-time Services Library (**librts.a**)

Syntax

```
#include <vrcppr.h>

int vrcppr (request, path)
char request, *path;
```

Description

The **vrcppr** subroutine issues a **DEFINE_CODE** supervisor call to add or delete a protocol procedure from Virtual Resource Manager (VRM). A **protocol procedure** is a special-purpose program that runs in VRM to support an application or service running on AIX.

The calling process's effective user ID must be superuser to use the **vrcppr** subroutine.

The value of the *request* parameter determines whether to add or delete the protocol procedure:

- 'a' Adds the protocol procedure that is contained in the file named by the *path* parameter. The file must be an executable file in **a.out** format.
- 'd' Deletes the protocol procedure named by the *path* parameter.

In both cases the *path* parameter must specify the full path name.

The **/etc/system** and **/etc/master** files must be set up correctly for the **vrcppr** subroutine to work properly. The **/etc/system** file must contain a stanza with the following information:

<i>sstname</i> :		The name of this /etc/system stanza.
protocol	= true	This is a protocol procedure.
nospecial	= true	No /dev special file is to be created.
noshow	= true	The devices command should not display this device
driver	= <i>mstname</i>	The name of the corresponding stanza in /etc/master .

The **/etc/master** file must contain a corresponding stanza with the following information:

<i>mstname</i> :		The name of this /etc/master stanza.
protocol	= true	This is a protocol procedure.
code	= <i>path</i>	The full path name of the protocol procedure.
iocn	= <i>iocn</i>	The IOCEN for this code.

The value of the **code** keyword must be the same as the *path* parameter passed to the **vrcppr** subroutine.

Return Value

Upon successful completion, a nonnegative integer value representing the IOCEN of the installed protocol procedure is returned. If the **vrcppr** subroutine fails, then it returns one of the following negative values:

VRM_inaa	The calling process's effective user ID is not superuser.
VRM_pnnf	The file named by the <i>path</i> parameter does not appear as the value of a code keyword in the /etc/master file.
VRM_snnf	The file named by the <i>path</i> parameter does not appear as the name of stanza in the /etc/system file.
VRM_mopn	The /etc/master file cannot be opened.
VRM_sopn	The /etc/system file cannot be opened.
VRM_defpp	The vrconfig program failed.
VRM_bopt	The <i>request</i> parameter is not 'a' or 'd'.
VRM_fork	The fork system call failed.
VRM_pnam	The <i>path</i> parameter does not specify a valid full path name.
VRM_iocn	The stanza of the /etc/master file that contains code = <i>path</i> does not contain definition for the iocn keyword.
FORKERR	The fork system call failed.
EXECERR	The exec system call failed.

Files

/etc/master
/etc/system
/etc/vrmconfig

Related Information

In this book: “attributes” on page 4-20, “config” on page 6-7, “master” on page 4-93, and “system” on page 4-133.

